情報処理学会論文誌　コンピューティングシステム　Vol. 3　No. 1　50–60 (Mar. 2010)

ファイル操作のシステムコール発行頻度に基づくバッファキャッシュ制御法の提案

片　上　達　也[†1]　　田　端　利　宏[†1]　　谷　口　秀　夫[†1]

バッファキャッシュは，ディスク装置に格納されたデータの入出力処理を高速化するために実装されている．バッファキャッシュの制御方式として，現在までに多くの手法が提案されたものの，いまだに多くのオペレーティングシステムでは LRU 方式が利用されている．一方，応用プログラムの動作内容をバッファキャッシュに反映するには，応用プログラムの発行したファイル操作のシステムコールに着目するのが良いと考えられる．そこで，本論文では，ファイル操作のシステムコール発行頻度に基づいて，バッファキャッシュを制御する手法を提案する．提案手法は，ファイル操作情報を基にファイルの重要度を決定し，重要度に基づいてバッファキャッシュのブロック置き換えを行う．本論文では，応用プログラムによる評価から，提案手法が入出力性能を向上させることを示す．

Proposal of I/O Buffer Cache Mechanism Based on the Frequency of System Call of the File Operation

Tatsuya Katakami,[†1] Toshihiro Tabata[†1]
and Hideo Taniguchi[†1]

Buffer cache is implemented to improve I/O performance with data in disks. As buffer cache management, there are many mechanisms, but still many operating systems deploy LRU (Least Recently Used) algorithm. On the other hand, to reflect process contents of application programs to buffer cache, management scheme based on the system calls which application programs requested is better. Then, we propose I/O buffer cache mechanism based on the frequency of system call of the file operation. Our proposed mechanism calculates file importance from information of file operation. In addition in buffer cache blocks are replaced based on this file importance. In this paper, we describe our mechanism improves I/O performance by evaluation of application programs.

1. はじめに

ディスク装置に格納されたデータの入出力処理を高速化するため，バッファキャッシュが多くのオペレーティングシステム（以降，OS と略す）で用いられている．バッファキャッシュ制御方式は，ブロックと呼ばれる固定長のデータを管理し，ブロック単位でバッファキャッシュのデータを置き換える．このため，キャッシュヒット率を向上させるためには，バッファキャッシュのブロック置換規則が重要になる．

ブロック置換規則には，ブロックに関する情報を基に制御する方式とファイルに関する情報を基に制御する方式がある．ブロックに関する情報を基に制御する代表的な手法として，LRU 方式がある．しかし，LRU 方式では，1 つの大きなファイルの入出力によって，他のファイルがバッファキャッシュから解放されてしまい，キャッシュヒット率が低下するという問題がある．また，特定のファイル群に対して繰り返しアクセスするときに，そのデータ量がバッファキャッシュの大きさを上回るとキャッシュヒット率が低下する．このような問題が発生するアクセスパターンを持つ事例として，Web サーバへのアクセス，および DBMS へのアクセスがある．

これらの問題を解決するため，ブロックアクセスのより詳細な情報を利用する手法が提案されている．たとえば，アクセス頻度[1),2)]，アクセス間隔[3),4)]，およびブロックアクセスの周期性[5),6)] の情報を利用する手法がある．しかし，DBMS などの特定の応用プログラム（以降，AP と略す）を指向した手法では，他の AP のファイルの入出力の性能を低下させるという問題がある．また，処理の優先度をバッファキャッシュ制御に反映できないという問題がある．さらに，ブロックアクセスごとに情報を取得するため，その制御オーバヘッドの増大と収集する情報量の増加という問題がある．

これに対して，ファイルの情報に基づくバッファキャッシュ制御方式が提案された．UBM 方式[7)] は，ファイル単位のアクセスの特徴を属性として取得し，この属性に基づきバッファキャッシュを制御する手法で，ブロック制御のたびにファイルのアクセス属性を取得する．このため，処理が複雑でオーバヘッドが大きいという問題がある．ディレクトリ優先方式[8)] は，利用者が優先して保護するファイルが存在するディレクトリを指定することで，指定し

†1 岡山大学大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama University

© 2010 Information Processing Society of Japan

たディレクトリ下のファイルを保護する．このため，あらかじめ使用するファイルが分かっていないと利用が難しい．

APがファイルを操作するのはシステムコール発行時である．このため，ファイル操作情報の取得をブロック制御時ではなく，ファイルを操作するシステムコールの発行時とすることで，オーバヘッドの削減が期待できる．また，利用するファイルを操作した情報（以降，ファイル操作情報と略す）をシステムコール発行時に自動取得することにより，APがどのファイルを扱うかを利用者が意識する必要がなくなる．

そこで，本論文では，ファイル操作のシステムコール発行頻度に基づくバッファキャッシュ制御法を提案する．提案手法は，ファイルを操作するシステムコールの発行を契機として，ファイル操作情報を取得する．また，一定の周期でファイル操作情報を集計し，ファイルの重要度を算出し，この重要度に基づき，2レベルに分割したバッファキャッシュを制御する．これにより，OSにおけるバッファキャッシュ制御に実行中のAPの操作内容を反映させ，キャッシュヒット率を向上させることを目指す．また，提案方式を実装し，評価した結果を報告する．

2. ファイル操作のシステムコール発行頻度に基づくバッファキャッシュ制御法

2.1 設　計

2.1.1 設計方針

提案方式の設計方針を以下に述べる．

(1)　ブロックへのアクセスごとにではなく，ファイルへの操作を基本とする単位でファイル操作情報を取得し，バッファキャッシュのバッファ置き換えの判断と制御にこの情報を利用する．ブロックよりも管理する数や操作回数が少ないファイルを利用することで，制御のために管理する情報量を削減する．

(2)　重要度に基づく2レベルのバッファキャッシュ方式を採用する．これにより，重要度に基づく制御とLRU制御の両者の特徴を生かした制御を可能にする．

(3)　ファイルごとにバッファキャッシュに格納される優先度（以降，重要度と呼ぶ）を決定する．重要度を決定するために，ファイルの操作要求（openシステムコール）時と操作終了（closeシステムコール）時に操作内容を取得する．readやwriteシステムコール発行時におけるバッファキャッシュ領域のバッファ置き換え時には，重要でないファイルに属するブロックを置き換え対象とする．これにより，APが意識するファイルの情報を基にバッファキャッシュを制御し，キャッシュヒット率向上と処理の高速化を目指す．

2.1.2 利用事例

提案手法は，ファイルごとに優先度を決定し，バッファキャッシュを制御するという特徴を持つ．以下に提案手法の有効な利用事例を示す．

(1)　Webサーバ処理

Webサーバ処理は，ファイルに対するアクセスのほとんどがシーケンシャルアクセスである．Webサーバ処理において，ファイルの使用頻度がブロックの使用頻度とほぼ等しいため，重要度の精度が向上し，キャッシュヒット率を向上させ，Webサーバの応答時間を短縮することができると見込まれる．

(2)　バックアップ処理中の他サービス

バックアップ処理は，バックアップ対象のファイル群に対してシーケンシャルにアクセスを行う．LRU方式では，サービス運用中にバックアップ処理を行うと，キャッシュヒット率を大きく低下させる．この事例として，Webサーバ運用中のバックアップ処理がある．提案手法において，バックアップ処理だけのためにオープンされたファイルが重要と判断されることはなく，それまで頻繁にアクセスされていたファイルを保護することができる．

(3)　コンパイル処理

コンパイル処理では，コンパイル対象のソースコードファイルに対してシーケンシャルアクセスを行い，コンパイル処理を行うために必要なライブラリに対してランダムアクセスを行う．このため，ファイルに対してシーケンシャルアクセスを行う場合は，ファイルの重要度に基づくバッファキャッシュ制御による性能向上によって処理時間の短縮が見込まれる．

2.1.3 基本方式

提案手法の基本方式を**図1**に示し，以下に述べる．提案手法には，2つの処理契機がある．1つは，ファイル操作時で，もう1つは，データ入出力時である．

また，ファイル操作時には，ファイル情報表にファイルごとの操作情報を格納する．さらに，ファイル情報表に一定回数以上の情報がたまると，ファイルごとの重要度を再計算し，その情報を重要度表に格納する．ファイル情報表と重要度表の内容を**表1**と**表2**に示す．ファイルはinode番号で識別し，ファイル操作情報をファイル情報表に格納し，ファイル操作情報から算出した重要度を重要度表に格納する．

提案手法では，バッファキャッシュのために確保された領域を2つのキューに分けて，キャッシュしたブロックを管理する．1つは，重要と判断されたファイルを構成するブロックが格納されたキューである．このキューに格納されたブロック群を重要ブロック群と呼ぶ．もう1つは，重要と判断されなかったファイルを構成するブロックが格納されたキューである．

© 2010 Information Processing Society of Japan

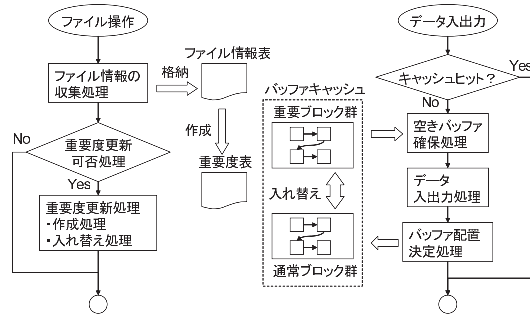


図 1　基本方式
Fig. 1　Design of the proposed I/O buffer cache mechanism.

表 1　ファイル情報表
Table 1　File information table.

項目	内容
inode 番号	ファイルの識別情報
オープン回数	更新前のオープン回数に重み（2.2.4 項で述べる）を乗じた数と最後の更新契機からのオープン回数の和
ファイルサイズ	ファイルの大きさ
最新オープン時刻	最後にオープンされた時刻

表 2　重要度表
Table 2　Importance table.

項目	内容
inode 番号	ファイルの識別情報
重要度	バッファキャッシュに格納される優先度

このキューに格納されたブロック群を通常ブロック群と呼ぶ．以降では，各キューにブロックを追加する処理を格納，一方のキューから取り出し，他方のキューにブロックを追加する処理を移動と呼ぶ．

ファイル操作時（open または close システムコール発行時）の処理の流れを以下に述べる．

(1)　open と close のシステムコールからファイル操作情報を取得し，ファイル情報表に格納する．

(2)　重要度更新処理を行うか否か判断する．重要度を更新する契機であれば重要度更新処理を行い，契機でなければ処理を終了する．

(3)　重要度更新処理を行う場合，ファイル操作情報から重要度を算出し，重要度表を作成する．次に，重要度更新処理後に，新たに重要と判断したファイルに属するブロックをバッファキャッシュの通常ブロック群から重要ブロック群に移動する．また，重要でなくなったファイルに属するブロックを重要ブロック群から通常ブロック群に移動する（入れ替え処理）．

データ入出力時の処理の流れを以下に述べる．

(1)　read または write システムコールを契機として，要求されたデータがキャッシュヒットしたか否かを調べる．

(2)　キャッシュミスした場合，バッファキャッシュから空きバッファを確保する．

(3)　データ入出力処理を行う．

(4)　データ入出力処理でアクセスしたブロックが構成するファイルが重要であれば重要ブロック群に，重要でなければ通常ブロック群に格納する．なお，重要かどうかの判断には，保護ファイル数という閾値を用い，重要度の順番で上位から保護ファイル数分だけのファイルを重要と判断する．

2.2　実　　装

2.2.1　課　　題

提案手法の実装における課題を以下にあげる．

(1)　収集するファイル操作情報
重要度を適切に算出するために必要なファイル操作情報を収集する．

(2)　重要度の算出方法
AP に利用されているファイルをバッファキャッシュに保護できるように，重要度の計算式を決定する．

(3)　重要度を更新する契機
実行中の AP のファイル利用に合わせて重要度を更新するため，重要度を更新する契機を決定する．

(4)　バッファキャッシュの入れ替え処理
重要度の算出後，これまで重要であったファイルが重要でなくなり，重要でなかったファイルが重要になることがある．これに合わせて，バッファキャッシュを操作する．

(5)　空きバッファ確保処理
通常ブロック群にバッファがなくなったときの空きバッファ確保処理を検討する．

© 2010 Information Processing Society of Japan

2.2.2　収集するファイル操作情報

重要度には，以下の 2 つの目標がある．

(1)　再利用される可能性の高いファイルほど重要度が高くなること

(2)　実行中の AP 群に合わせて更新できること

これらの目標を達成するためには，ファイル利用状況を示す情報を収集し，かつ現在またはこれからのアクセスに応じて，重要度を高くすることが望ましい．そこで，ファイルの利用状況を示す参照数とオープン回数を用いることが考えられる．また，バッファキャッシュは有限であることから，特定のファイルだけをバッファキャッシュに格納することは，他のファイルのキャッシュヒット率を低下させるため，望ましくない．そこで，ファイルサイズを用いることが考えられる．

参照数とは，ファイルを同時にオープンしているプロセスの数である．AP は，AP の利用する仮想記憶空間上に 1 度読み込んだファイルのデータを蓄えておくことで，ディスク装置との入出力を省略する．このため，現在多くの AP から参照されているファイルが，現在オープンしている AP に今後も利用されるとは必ずしもいえないため，重要度計算に参照数を利用しないこととした．なお，実験により，参照数を用いる効果が非常に小さいことを確認している．

オープン回数とは，open システムコールの発行回数である．頻繁にアクセスされるファイルは，過去にオープンされた回数も大きく，将来も頻繁に使用されると予測できる．たとえば，Web サーバ処理とコンパイル処理では，繰り返しオープンされるファイルは，頻繁に再利用されていることが分かっている．

ファイルサイズとは，オープンまたはクローズしたファイルの大きさである．単一のファイル，または 1 度しかアクセスされないファイル群へのシーケンシャルアクセスによってバッファキャッシュが占有されることを防ぐため，サイズを考慮し，格納領域を選択する．なお，ファイルサイズが大きいほど，ランダムアクセスが発生する可能性が高いため[9]，サイズが大きいファイルの頻繁にアクセスされるブロックを通常ブロック群で管理しても，そのブロックは通常ブロック群に残る可能性が高く，キャッシュヒット率の低下を抑制することができると考えられる．

上記により，重要度計算に必要な情報として，オープン回数とファイルサイズをファイル情報表に格納する．また，これらに加えて，ファイルを特定するために，i ノード番号が必要である．さらに，2 つのファイルの重要度が等しい場合に順位付けをするために，最近 open システムコールが発行された時刻として，最新オープン時刻が必要である．

2.2.3　重要度の算出方法

2.2.2 項で述べたように，重要度はオープン回数を基にして算出する．また，オープン回数が多くても，一定のサイズより大きいものを通常ブロック群で管理する．これらに基づき，決定した重要度の計算式を式 (1) に示す．

$$
\begin{aligned}
&if\ ((O_{cnt} > O_{min})\&\&(F_{size} < F_{sizemax}))\{\\
&\quad Importance = O_{cnt};\\
&\}\ else\{\\
&\quad Importance = FALSE(=0);\\
&\}
\end{aligned}
\tag{1}
$$

式 (1) において，O_{cnt} は計算対象ファイルのオープン回数，F_{size} はそのファイルサイズ，O_{min} と $F_{sizemax}$ は閾値，$Importance$ は重要度を示す．

この計算式によって，ファイルサイズが $F_{sizemax}$ より小さく頻繁にオープンされるファイルに属するブロックを重要ブロック群に格納し，キャッシュヒット率を向上させる．O_{min} を大きくすることで，非常に頻繁にオープンされるファイルだけを重要ブロック群に格納できる．なお，2.2.4 項で述べる重みを 1 に近い値にした場合，または更新契機を長くした場合には，オープン回数が大きくなるため，O_{min} の値により，重要ブロック群に格納できるファイル数が大きく変化する．また，サイズの大きいファイルや頻繁にオープンされないファイルを通常ブロック群で管理することにより，1 つのファイルまたは特定のファイル群に重要ブロック群を占有されることを防ぐ．

2.2.4　重要度を更新する契機

open，close システムコールの発行状況は，つねに変動するため，重要度表の更新契機は，ファイルのオープンとクローズの操作回数に連動するのが望ましい．このため，重要度表の更新契機は，前回の重要度表の更新から一定回数のオープンまたはクローズが行われたときとする．

しかし，ファイル情報表の大きさを固定とした場合，一定回数のオープンまたはクローズが行われる前に，ファイル情報表があふれることがある．このため，重要度表の更新契機として，さらに，ファイル情報表に一定数のファイル操作情報が蓄積されたときも採用する．

また，ブロックの使用頻度に基づく LFU 方式は，多くの処理において，キャッシュヒット率が低い[10),11)]．これは，LFU 方式が，時間局所性を考慮せず，時間が経過すると使用頻度が増加し続けるため，最近使用したブロックに対してキャッシュヒット率が低いことが原因である．提案手法においても，単にファイルの使用頻度（オープン回数）を加算していく

© 2010 Information Processing Society of Japan

だけでは，古い情報ほど重要度が高くなる可能性が高い．

よって，LRFU 方式[2] のように，重要度表の更新時に，全ファイル操作情報のオープン回数に重み w をかけることで，古い情報の影響を周期的に小さくする．これにより，最近，頻繁にオープンされたファイルの重要度が高くなり，時間局所性を持った多くのアクセスパターンにおいて，キャッシュヒット率を向上させることが期待できる．

2.2.5　バッファキャッシュの入れ替え処理

重要度を算出したときに，新しく算出した重要度に合わせて，バッファキャッシュの格納領域を入れ替える．これによって，重要度の高いファイルに属するブロックが通常ブロック群に格納されること，および重要度の低いファイルに属するブロックが重要ブロック群に格納されることを防ぎ，キャッシュヒット率の低下を抑制する．

2.2.6　空きバッファ確保処理

空きバッファの確保は，通常ブロック群から LRU 方式でブロックを選んで解放することで行う．もし通常ブロック群に解放可能なブロックがないとき，重要ブロック群内で最も重要度の低いファイルを構成するブロックすべてを重要ブロック群から通常ブロック群に移動し，通常ブロック群から LRU 方式でブロックを選んで解放する．

2.3　期待される効果

提案手法によって，以下のことが期待できる．

(1)　提案手法では，一定サイズ未満のファイルはファイルごとの重要度で管理する．これにより，ファイルの重要度に基づくバッファキャッシュ制御によって，キャッシュヒット率の向上が見込まれる．

(2)　提案手法は一定サイズ以上の大きなファイルに属するブロックが重要ブロック群に格納されない．これによって，1 つの大きなファイルに対するシーケンシャルアクセスが発生した場合にも，重要なファイルが保護され，他サービスへの影響を抑制することができる．

(3)　open と close システムコールの発行回数は，ブロックの制御回数に比べて少ない．このため，UBM 方式のように，ブロック制御時にファイルの情報を取得する手法に比べて，オーバヘッドと制御に用いるデータの情報量を削減できる．

3．評　　価

3.1　評価の観点

マイクロベンチマークの評価では，頻繁に発生するファイルのアクセスパターンにおける提案手法の性能を評価する．また，マクロベンチマークの評価では，これらのアクセスパターンが現れる実 AP における提案手法の性能を評価する．

3.2　マイクロベンチマーク

3.2.1　評 価 環 境

提案手法（以降，図表では Frequency of File Usage：FFU 方式と略す）を FreeBSD 6.3-RELEASE（以降，FreeBSD 6.3-R と略す）に実装し，評価した．評価では，1 台の計算機（CPU：Celeron 430（2.4 GHz），メモリ：256 MB，OS：FreeBSD 6.3-R，バッファキャッシュサイズ：128 MB，VMIO：有効，バッファの大きさ：16 KB）を利用した．なお，FreeBSD 6.3-R に元から実現されている LRU 方式を比較対象とした．

3.2.2　設定したパラメータ

重要度表の大きさは，各実験において利用するファイルをすべて格納できる数（512 個）とした．また，O_{min} を 0 に設定し，重要度が 0 より大きいファイルのブロックを重要ブロック群に格納する．ファイルサイズ制限（$F_{sizemax}$）は 2 MB とした．重みは，最大値 1 と最小値 0 の中間の値である 0.5 とした．

3.2.3　アクセスパターンの分類と評価

ファイルのアクセスパターンとして，シーケンシャルアクセス，ループアクセス（特定のデータ群に対して，周期的な規則性に基づき繰り返しアクセスを行うこと），およびランダムアクセスの 3 つ[7] について，実験した．

シーケンシャルアクセスは，以下の処理で実験を行った．1 MB のファイル 10 個に対してランダムに 200 回アクセスし，各ファイルをシーケンシャルに読み込んだ後，1 GB のファイルをシーケンシャルに読み込む．その直後に，前に読み込んだものと同じ 1 MB のファイル 10 個を 1 回ずつシーケンシャルに読み込む．ここでは，保護するファイルの数を 10 個とし，重要度の更新契機を 200 とした．評価結果から，LRU 方式では 1 GB ファイルへのアクセスにより，キャッシュがすべて置き換えられ，キャッシュヒット率が 0%になった．一方，提案手法ではキャッシュヒット率低下を抑制でき，処理時間を短縮できた．

ループアクセスは，1 MB のファイル 256 個に対して，シーケンシャルに 10 回繰り返して読み込む処理の処理時間を測定した．ここでは，保護ファイル数を 100，重要度表の更新契機を 768 とした．1 回のループアクセスでアクセスするデータサイズがキャッシュサイズよりも大きいため，LRU 方式ではキャッシュヒット率が 0%となった．一方，提案手法は一部のファイルを保護できたため，キャッシュヒット率は 15.6%となり，処理時間も短縮できた．

ランダムアクセスは，1 GB のファイル 1 個に対して，100,000 回 open，lseek，read，close を繰り返す処理の処理時間を測定した．lseek で動かすヘッドの位置をランダムにし

© 2010 Information Processing Society of Japan

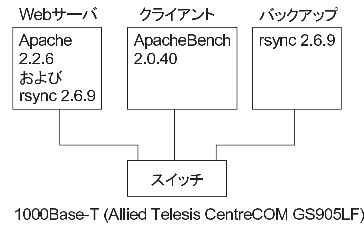


図 2　Web サーバ実験の計算機構成
Fig. 2　Environment of Web server evaluation.

て，100,000 回のランダムアクセスを行い，16 KB ずつ read システムコールを発行した．保護ファイル数は 1 とし，重要度表の更新契機を 100 とした．評価結果から，LRU 方式と提案手法は，ほぼ等しい処理時間，およびキャッシュヒット率となった．この場合，提案手法は，アクセス対象の 1 GB のファイルを保護しないため，LRU 方式と同等の性能となる．また，提案手法のオーバヘッドにより，処理時間が低下していないことが分かった．なお，この評価のアクセスパターンは，DBMS のアクセスパターンに近いため，DBMS においても，提案手法は LRU 方式と同等の性能になると考えられる．

3.3　マクロベンチマーク

3.3.1　評 価 環 境

評価では，3 台の計算機（CPU：Celeron 430（2.4 GHz），メモリ：256 MB，OS：FreeBSD 6.3-R，VMIO：有効）を利用した．Web サーバ処理で利用するファイルの総データサイズが約 520 MB であったため，メモリサイズを 256 MB とし，総データサイズよりも小さい値に設定した．これは，バッファキャッシュの置き換えを発生させることによって，ブロック置換アルゴリズムの性能を比較評価するためである．また，バッファキャッシュサイズは，メモリスラッシングが発生しない最大の値として，128 MB に設定し，バッファの大きさは FreeBSD 6.3-R のデフォルトである 16 KB とした．なお，FreeBSD 6.3-R に元から実現されている LRU 方式を比較対象とした．

Web サーバ実験での計算機の構成を**図 2** に示す．Web サーバとして Apache 2.2.6，バックアップ用 AP として rsync 2.6.9 を利用した．Web クライアント機，Web サーバ機，およびバックアップ用計算機は，1000Base-T のスイッチを介して接続されている．

3.3.2　設定したパラメータ

表 3 に評価で利用したパラメータを示し，その内容について述べる．重要度表の大きさ（重要度表のエントリ数）は，評価で利用する約 60,000 のファイルすべてを格納できる値として，65,536 個とした．

表 3　Web サーバ処理とバックアップ処理の評価における各パラメータ
Table 3　Parameters of evaluation of Web server process and backup process.

通番	パラメータ名	値
(1)	重要度表の大きさ	65,536 個
(2)	保護ファイル数	3,200 個
(3)	重要度表の更新契機	128,000
(4)	O_{min}	0
(5)	ファイルサイズ制限（$F_{sizemax}$）	2 MB
(6)	重み	0.5

保護ファイル数は，VMIO 領域も含めたキャッシュサイズ（200 MB）をリクエストしたファイルの平均サイズ（61 KB）で割った値（3,200 個）とした．これは，キャッシュサイズに対して，格納されるファイルのおおよその数を算出し，この数を目安に保護するファイル数を決定するためである．これにより，重要ブロック群に入りきらないファイルまで重要と判断することがなくなり，重要度更新処理後の入れ替え処理のオーバヘッドを削減できる．なお，保護ファイル数が小さすぎると，本来重要であるべきファイルが重要と判断されず，重要ブロック群を有効に使えないという問題が発生する．

重要度表の更新契機は，重要度表の更新後，つねに 3,200 個以上のファイルが重要度表に残存する値として，保護ファイル数の 40 倍とした．これは，Web サーバ処理において，重要度表の更新後に，保護ファイル数以上のファイルを重要度表に残すことができる値である．多くの場合で，40 倍が十分な数のファイル操作情報を重要度表に残し，性能が良い値である結果を実験で得ている．なお，オープン回数が 0 になったファイルの操作情報は，重要度表の更新時に重要度表から削除される．

重要度表の更新契機は，小さすぎると，オープン回数が 1 以上のファイルが少なくなり，設定した保護ファイル数以上のファイル操作情報が重要度表に残らないため，キャッシュヒット率を低下させる．また，大きすぎると，アクセスパターンの急激な変化に対応することができず，キャッシュヒット率が低下するという特徴がある．

O_{min} を 0 に設定し，重要度が 0 より大きいファイルのブロックを重要ブロック群に格納する．これは，たとえば O_{min} を 1 にすると重要と判断されるファイルがあまり存在しないためである．

ファイルサイズ制限（$F_{sizemax}$）は，頻繁に（50 回以上）読み込まれるファイルの最大

© 2010 Information Processing Society of Japan

ファイルサイズより大きい値とした．

重みは，最大値 1 と最小値 0 の中間の値である 0.5 とした．これは次の理由による．重要度表の更新契機が長くなると，古い情報が重要度表に残りやすくなるので，重みを小さく設定し，古い情報の影響を小さくする必要がある．一方，重要度表の更新契機が短い場合，新しい情報まで重要度表から削除する可能性があるので，重みを大きく設定し，古い情報の影響を大きくする必要がある．

3.3.3　Web サーバ処理とバックアップ処理

ファイルのシーケンシャルアクセスがきわめて頻発する事例である Web サーバ処理，およびバックアップ処理において，提案手法が有効に動作することを示す．

Web サーバ処理では，岡山大学の Web サーバへのリクエストパターンから抽出した 100,000 回のリクエストに対して評価を行った．また，評価の前に同じ 100,000 回のリクエストを要求することで，キャッシュを定常状態とした．評価における各ファイルへのアクセス回数を累積確率で**図 3** に示す．約 56%のファイルへのアクセスは 200 回以下であり，700 回から 800 回のアクセスのファイルが約 20%，800 回から 1,000 回のアクセスのファイルが約 20%あることが分かる．

バックアップ処理は rsync によって行った．バックアップ処理中の Web サーバの平均応答時間は，提案手法と LRU 方式において，ともに rsync が動作している時間として，rsync 起動後 50,000 回の平均応答時間を測定した．また，Web サーバ処理の性能向上によって，バックアップ処理に与える影響を調査するために，バックアップ処理の処理時間を測定した．Web サーバ処理で要求するファイルは，約 16,000 ファイル（約 520 MB）で，バックアップ処理で利用するファイルは岡山大学の Web サーバにある約 60,000 ファイル（約 6 GB）である．

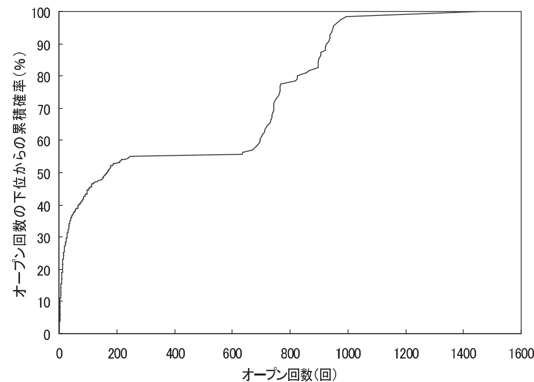


図 3　評価におけるファイルごとのアクセス回数
Fig. 3　Number of access of each file on the evaluation.

提案手法を FreeBSD 6.3-R に実装し，Web サーバ処理の平均応答時間を測定した．**表 4** に，Web サーバの平均応答時間を示す．表 4 より，提案手法は，従来方式に比べて，平均応答時間を 11.6%短縮することが分かる．これより，提案手法は，Web サーバ処理において有効なバッファキャッシュ制御が行えているといえる．また，バックアップ処理を行っている際には，提案手法は，従来方式に比べて，平均応答時間を 32.9%短縮する．これより，提案手法は，バックアップ処理による Web サーバ処理への影響を抑制しているといえる．

また，**表 5** に，rsync によるバックアップの処理時間を示す．表 5 より，提案手法では，Web サーバ処理の性能を向上させたことで，バックアップ処理の処理時間が長くなることはないといえる．むしろ，ここでは，処理時間を 17.5%短縮している．これは，Web サーバで要求したファイルとバックアップ処理で利用するファイルが重複しているために，提案手法では，重複したファイルの中で頻繁に読み込まれるファイルについてキャッシュヒット率が向上していることが原因である．このため，提案手法は，Web サーバ処理，およびバックアップ処理中の Web サーバ処理に対して，有効であるといえる．

3.3.4　シミュレータによるキャッシュヒット率とオーバヘッドの評価

バッファキャッシュのシミュレータを作成し，これに LRU 方式，LRFU 方式，および提案手法（FFU 方式）を実装し，岡山大学の Web サーバから抽出した 100,000 回のファイ

表 4　Web サーバの平均応答時間
Table 4　Average response time of Web server.

測定項目	平均応答時間（応答時間比）			
	バックアップなし		バックアップあり	
LRU 方式	3.13 ms	（1.00）	9.56 ms	（3.05）
FFU 方式	2.77 ms	（0.88）	6.41 ms	（2.05）

表 5　rsync によるバックアップの処理時間
Table 5　Execution time of rsync backup process.

測定項目	処理時間（処理時間比）			
	バックアップのみ		Web サーバと同時	
LRU 方式	950 sec	（1.00）	1,346 sec	（1.42）
FFU 方式	950 sec	（1.00）	1,111 sec	（1.17）

© 2010 Information Processing Society of Japan

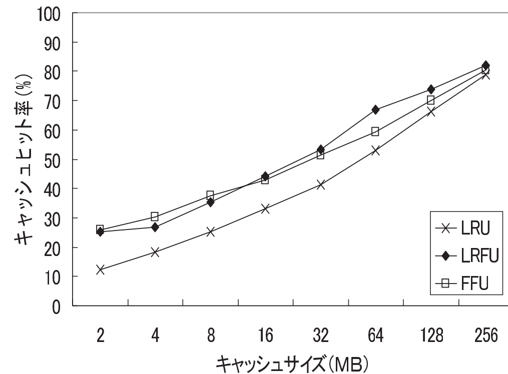


図 4　シミュレータによる Web サーバ処理のキャッシュヒット率
Fig. 4　Cache hit rates of Web server process by simulator.

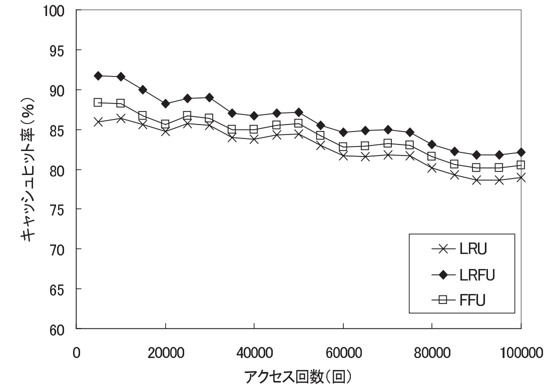


図 5　アクセス回数と Web サーバ処理のキャッシュヒット率
Fig. 5　Relation between access and cache hit rates of Web server on the simulator.

ル入出力を与えて，バックアップ処理なしの場合のキャッシュヒット率を評価した．LRFU 方式の管理表の大きさは，処理中のブロックの全情報を格納可能な数として，約 390,000 エントリ，重み付けを行う契機は，提案手法と等しい値として 128,000 回のブロックアクセスごととした．

FreeBSD 上で利用できるキャッシュメモリには，バッファキャッシュ領域に加えて，VMIO 機能で利用できるページキャッシュ領域もある．実際のページキャッシュ領域の大きさは，プログラムの実行状況に応じて変化する．この変化の再現は難しいため，このシミュレータによる評価では，バッファキャッシュ領域とページキャッシュ領域を合わせたキャッシュサイズを固定としてシミュレーションした．

図 4 にシミュレータによる評価でのキャッシュサイズとキャッシュヒット率の関係を示す．図 4 から，提案手法は，LRU 方式と比較してキャッシュヒット率を向上させ，LRFU 方式と比較して，ほぼ同等のキャッシュヒット率であるといえる．LRU 方式に対しては，キャッシュサイズが小さいほど，大きくキャッシュヒット率が向上する傾向がある．特にキャッシュサイズが 64 MB 以下では，LRU 方式の半分のキャッシュサイズで同等以上のキャッシュヒット率を実現している．また，LRFU 方式に対しては，キャッシュメモリが小さいほど性能が向上する傾向があるものの，キャッシュメモリが大きい場合は，キャッシュヒット率が下回る傾向がある．

キャッシュサイズが 256 MB の場合について，アクセス回数とキャッシュヒット率の関係を**図 5** に示す．これは，アクセス 5,000 回ごとにキャッシュヒット率を計算したものである．各方式ともアクセス回数が増えるにつれて，新たなファイルにアクセスすることがあるため，キャッシュヒット率が低下している．また，各方式間でキャッシュヒット率が逆転することはなく，キャッシュヒット率が推移していることが分かる．

次に，各方式で利用する管理表の大きさと更新契機について述べる．評価では，LRFU 方式の管理表をすべてのブロックに対するエントリが格納できる大きさとした．このため，LRFU 方式のキャッシュヒット率は，大幅に向上している．その一方で，提案方式（約 1.75 MB, 65,536 エントリ）に比べて，LRFU 方式で制御に用いる管理表（約 6.0 MB，約 390,000 エントリ）は大きい．これは，LRFU 方式の持つ管理表の大きさは，処理に必要なブロックの数に比例するのに対して，提案手法では，操作するファイル数に比例するためである．このため，LRFU 方式では，処理に必要なファイルの平均サイズがブロックサイズよりも比較的大きい場合には，各ファイルを構成するブロックの数が増加するため，管理表を大きくする必要がある．たとえば，4 KB のブロックで 1 GB のファイル入出力があった場合，256K 個のエントリを持つ必要がある．一方，提案手法では，1 GB のファイルに対する 1 つのエントリを持つだけでよい．また，管理表が大きいため，必要なエントリの検索にコストがかかる．

管理表の更新契機は，提案方式が open システムコールと close システムコールであるのに対し，LRFU 方式はブロックへの読み込み要求時となる．このため，更新頻度は提案方式に比べて非常に高い．評価では，提案方式の管理表の更新回数が約 200,000 回であるのに対

© 2010 Information Processing Society of Japan

表 6　シミュレータの実行時間
Table 6　Processing time of the simulator.

方式	処理時間 (s)
LRFU 方式	35.568
FFU 方式	32.775
LRU 方式	28.626

して，LRFU 方式は，約 840,000 回と 4 倍以上であった．このことから，実行時のメモリ消費量だけでなく，管理表更新のオーバヘッドについても LRFU 方式が大きいことが分かる．

参考値として，シミュレータでの各方式のシミュレーション時間を測定した．測定に用いた環境は，CPU：Core2 Duo 2.4 GHz，メモリ：2 GB，OS：Windows Vista である．測定結果を**表 6** に示す．シミュレータでは，FreeBSD に実現したのと同じバッファキャッシュ領域を実現し，シミュレーションしている．このため，シミュレータの実行時間から，実際のバッファキャッシュ領域操作と各方式の管理表の操作にかかるオーバヘッドを推察できる．表 6 から，LRFU 方式は管理表が大きく，管理表の更新頻度が高いため，実行時のオーバヘッドが最も高いと推察できる．提案手法（FFU 方式）のオーバヘッドは，キャッシュヒット率でほぼ同等の LRFU 方式より小さく，キャッシュヒット率で勝っている LRU 方式より大きいことが分かる．

以上のことから，提案手法はキャッシュヒット率で LRFU 方式に劣る場合があるものの，管理表が小さく，実行時のオーバヘッドが小さいといえる．LRU 方式に対しては，管理表や実行時のオーバヘッドが大きいものの，キャッシュヒット率が高く，特にキャッシュメモリが小さい場合の効果が大きいことが分かった．

3.3.5　カーネル make 処理による評価

FreeBSD 6.3-R のカーネル make 処理において提案手法を評価した．評価に用いたパラメータは，保護ファイル数と重要度表の更新契機以外は，表 3 と同じである．バッファキャッシュサイズ，保護ファイル数，および重要度表の更新契機を**表 7** に示す．保護ファイル数は，バッファキャッシュサイズをカーネル make 処理で利用する平均ファイルサイズで割った値とした．更新契機は，保護ファイル数の 40 倍の値とした．

実験は，カーネルが認識する主記憶の大きさを変えて行った．これにより，データ入出力量に対するキャッシュに利用できる主記憶のメモリが相対的に小さい場合の評価を行った．評価結果を**表 8** に示す．

表 7　測定のパラメータ
Table 7　Parameters used in the evaluation.

メモリサイズ	バッファキャッシュサイズ	保護ファイル数	更新契機
64 MB	12 MB	340	14,000
128 MB	22 MB	630	25,000
256 MB	34 MB	1,000	40,000

表 8　カーネル make 処理の評価結果
Table 8　Evaluation results of ‘make’ command of kernel.

メモリサイズ	キャッシュヒット率		処理時間	
	LRU 方式	FFU 方式	LRU 方式	FFU 方式
64 MB	98.80%	98.84%	949.02 s	940.25 s
128 MB	98.98%	99.00%	812.14 s	812.30 s
256 MB	99.41%	99.41%	790.16 s	791.30 s

評価結果から，提案手法のキャッシュヒット率は，LRU 方式と同等か，高いことが分かる．特にメモリサイズが小さくなるほど，LRU 方式に対してキャッシュヒット率が向上することが分かる．また，提案手法は，64 MB の場合には処理を高速化できているものの，制御オーバヘッドのため，128 MB と 256 MB の場合の処理が遅くなっていることが分かる．カーネル make 処理は，入出力処理だけでなく，演算処理の負荷も比較的高いため，キャッシュヒット率で勝っているものの処理が遅くなったものと推察できる．現在の提案手法の実装は，管理表の構造や処理方法が最適化されていないため，実装方法の見直しにより，オーバヘッドをより小さくすることが必要である．

4. 関 連 研 究

ファイルの情報に着目した手法として，UBM 方式[7] が提案されている．UBM 方式は，ブロックのアクセスパターンをファイルごとに sequential reference，looping reference，other reference（ランダムアクセス）のいずれかに分類し，それぞれの処理に合わせて，バッファキャッシュの制御方式を切り替える手法である．また，looping reference と other reference で置き換えるブロックを決定するために，ブロック置き換え後のキャッシュヒット率の見積りを行い，見積もられたキャッシュヒット率が小さい方のブロックを置き換える．UBM 方式では，ファイルに対して属性を付属させることによって，ファイルを意識したバッファキャッシュ制御を行い，複数 AP が並行して動作する場合にも高いキャッシュヒット率となっている．しかし，毎ブロックアクセス時にファイルの属性付けを行う必要があり，直接実装した

© 2010 Information Processing Society of Japan

場合のオーバヘッドが大きいと報告されている．

また，ディレクトリ優先方式[8]は，利用者が優先するディレクトリを指定し，指定したディレクトリ下のファイルを保護し，キャッシュヒット率を向上させる．しかし，優先するディレクトリを手動で決定する必要があるため，複数の AP が頻繁に切り替わり，使用するディレクトリが頻繁に切り替わる場合，AP の動作内容に合わせて優先ディレクトリを適切に決定することが難しい．

ファイルの特徴に基づくディスクキャッシュのヒット率見積りを用いた置換アルゴリズム[12]は，現在キャッシュしているブロックの量とファイルサイズから，置き換えることで最もキャッシュヒット率が低下しないと予測されるファイルのブロックを優先して置き換える．また，ファイルに対するアクセスの情報だけでなく，ファイルサイズという OS の保持している情報を利用し，各ファイルに対してランダムアクセスやシーケンシャルアクセスといった特徴的なアクセスが発生する場合に，キャッシュヒット率が高い．一方，一定期間内に複数のファイルに対して一様にランダムアクセスが発生したとき，または複数のファイルに対して一様にシーケンシャルアクセスが発生したとき，ファイルの特徴を特定できず，キャッシュヒット率を低下させるという問題がある．しかし，実 AP での評価は行われておらず，実 AP での有効性が明らかでない．

上記の UBM 方式とディレクトリ優先方式の問題は，AP の発行したシステムコールを契機としてファイルの情報を取得することで解決することができる．

5. おわりに

ファイル操作のシステムコール発行頻度に基づくバッファキャッシュ制御法を提案した．提案手法では，ファイル操作のシステムコール発行時に，ファイル操作情報を収集し，定期的に重要度を計算する．計算した重要度に基づきバッファキャッシュを制御することで，実 AP が操作するファイルの情報をバッファキャッシュ制御に反映する．

提案手法を FreeBSD 6.3-R に実装し，評価を行った．評価の結果，提案手法は，シーケンシャルアクセスとループアクセスにおいて LRU 方式よりも性能を向上させ，ランダムアクセスにおいて LRU 方式と同等の性能であることが分かった．

実 AP として，Web サーバ処理，およびバックアップ処理中の Web サーバ処理を評価した．評価の結果，LRU 方式と比較して，Web サーバ処理の平均応答時間を 11.6%，バックアップ処理中の Web サーバ処理の平均応答時間を 32.9%短縮した．なお，LRU 方式と比較して，Web サーバ処理中のバックアップ処理の処理時間を 17.5%短縮していることから，Web サーバ処理を優先することによってバックアップ処理の入出力性能が低下することを抑制しているといえる．シミュレータによる Web サーバ処理のキャッシュヒット率の評価では，提案手法は，LRFU 方式にキャッシュヒット率が劣る場合があるものの，管理表の大きさが小さく，その更新頻度が低いため，実行時のオーバヘッドが小さいことを示した．また，LRU 方式に対しては，オーバヘッドが大きいものの，キャッシュヒット率が高く，特にキャッシュメモリが小さい場合に有効であることが分かった．

カーネル make 処理による評価では，LRU 方式に対して，キャッシュヒット率は同等以上の性能でよいことが示せた．しかし，処理時間の面では，実行のオーバヘッドのため，LRU 方式よりも遅い場合があった．

今後の課題として，パラメータを自動的に決定する手法の検討がある．

謝辞　方式の検討にご協力いただいた岡山大学大学院自然科学研究科の乃村能成准教授に感謝します．

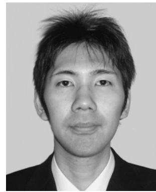
参　考　文　献

1) Robinson, J.T. and Devarakonda, M.V.: Data Cache Management Using Frequency-Based Replacement, *Proc. 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp.134–142 (1990).
2) Lee, D., Choi, J., Kim, J.H., Noh, S.H., Min, S.L., Cho, Y. and Kim, C.S.: LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies, *IEEE Trans. Comput.*, Vol.50, No.12, pp.1352–1361 (2001).
3) Phalke, V. and Gopinath, B.: An Inter-Reference Gap Model for Temporal Locality in Program Behavior, *Proc. 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp.291–300 (1995).
4) Jiang, S. and Zhang, X.: LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performancer, *Proc. 2002 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp.31–42 (2002).
5) Johnson, T. and Shasha, D.: 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm, *Proc. 20th International Conference on Very Large Databases*, pp.439–450 (1994).
6) Megiddo, N. and Modha, D.S.: ARC: A Self-Tuning, Low Overhead Replacement Cache, *Proc. 2nd USENIX Conference on File and Storage Technologies* (*FAST '03*), pp.115–130 (2003).
7) Kim, J.M., Choi, J., Kim, J. and Noh, S.H.: A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping Ref-

© 2010 Information Processing Society of Japan

erences, *Proc. 4th Symposium on Operating System Design and Implementation* (*OSDI 2000*), pp.119–134 (2000).
8) 田端利宏，小峠みゆき，乃村能成，谷口秀夫：ファイルの格納ディレクトリを考慮したバッファキャッシュ制御法の実現と評価，電子情報通信学会論文誌，Vol.J91-D, No.2, pp.435–448 (2008).
9) Roselli, D., Lorch, J.R. and Anderson, T.E.: A Comparison of File System Workloads, *Proc. 2000 USENIX Annual Technical Conference*, pp.41–54 (2000).
10) Tanenbaum, A.S.: *Modern Operating Systems*, 2nd Edition, Prentice Hall (2001).
11) Megiddo, N. and Modha, D.S.: Outperforming LRU with an Adaptive Replacement Cache Algorithm, *IEEE Computer Magazine*, pp.58–65 (2004).
12) 林　佳寛，鈴来和久，横田裕介，大久保英嗣：ファイルの特徴に基づくディスクキャッシュのヒット率の見積もりを用いた置換アルゴリズム，情報処理学会研究報告 2007-OS-104, Vol.2007, No.10, pp.95–102 (2007).

(平成 21 年 7 月 24 日受付)
(平成 21 年 12 月 24 日採録)



片上　達也（学生会員）
2008 年岡山大学工学部情報工学科卒業．同年同大学大学院自然科学研究科博士前期課程入学．現在，在学中．オペレーティングシステムに興味を持つ．

田端　利宏（正会員）
1998 年九州大学工学部情報工学科卒業．2000 年同大学大学院システム情報科学研究科修士課程修了．2002 年同大学院システム情報科学府博士後期課程修了．2001 年日本学術振興会特別研究員（DC2）．2002 年九州大学大学院システム情報科学研究院助手．2005 年岡山大学大学院自然科学研究科助教授．現在，同准教授．博士（工学）．オペレーティングシステム，コンピュータセキュリティに興味を持つ．電子情報通信学会，ACM，USENIX 各会員．

谷口　秀夫（正会員）
1978 年九州大学工学部電子工学科卒業．1980 年同大学大学院修士課程修了．同年日本電信電話公社電気通信研究所入所．1987 年同所主任研究員．1988 年 NTT データ通信株式会社開発本部移籍．1992 年同本部主幹技師．1993 年九州大学工学部助教授．2003 年岡山大学工学部教授．博士（工学）．オペレーティングシステム，実時間処理，分散処理に興味を持つ．著書『並列分散処理』（コロナ社）等．電子情報通信学会，ACM 各会員．

© 2010 Information Processing Society of Japan