ファイルの使用頻度に基づくバッファキャッシュ制御法

片 上 達 也† 田 端 利 宏†† 谷 口 秀 夫††
渡 辺 弘 志†† 乃 村 能 成††

既存の多くのオペレーティングシステムでは，入出力処理のバッファキャッシュをブロック単位で制御している．一方，応用プログラムは，ブロックではなくファイルのレベルで入出力を要求する．プロセッサ処理に比べて処理が遅い入出力処理を効率よく実行するためには，この違いをうまく考慮したバッファキャッシュ制御法が有効と考えられる．そこで，ここでは，応用プログラムのファイルの使用頻度に基づき優先して保持するブロックを決定するバッファキャッシュ制御法について述べる．

I/O Buffer Cache Mechanism Based on Use Frequency of File

Tatsuya KATAKAMI,† Toshihiro TABATA,††
Hideo TANIGUCHI,†† Hiroshi WATANABE††
and Yoshinari NOMURA††

Most operating systems manage a buffer cache for I/O processing by a block unit. On the other hand, application programs handle a data from a viewpoint of files. I/O buffer cache mechanism considered this difference is effective to execute I/O processing efficiently because I/O processing is slower than CPU processing. In this paper, we describe I/O buffer cache mechanism that decides blocks protected based on use frequency of file of application programs.

1. はじめに

ディスク装置とのデータ入出力において，バッファキャッシュ制御はオペレーティングシステム(以降，OSと略す)の重要な仕事である．既存の多くのOSは，バッファキャッシュを制御する単位をブロックとし，置き換えアルゴリズムとして固定長のブロックLRU方式を用いている．LRU方式は，頻繁にアクセスされるブロックをキャッシュに残すことができるため，キャッシュヒット率が高く，効率が良いことが知られている．

ブロック置き換えアルゴリズムの研究は，大きく3つの方式に分類される．1つ目は，各ブロックの参照順序や参照頻度に基づいてブロック置き換えを行う方式である．この方式の代表的な例として，LRU，FIFO，LFU，FBR[1]，LRU-k[2]，IRG[3]，Agingが存在する．2つ目は，利用者によってあらかじめ提供されたブロック参照パターンに基づいてブロック置き換えを行う方式である．この方式の代表的な例として，ACFC[4]，UBM[5]が存在する．3つ目は，ブロックの参照の規則性に基づいてブロック置き換えを行う方式である．この方式の代表例として，2Q[6]，SEQ[7]，EELRU[8]が存在する．これらの方式は，ブロックのアクセスに着目した方式であり，ファイルの情報は意識していない．

一方，応用プログラム（以降，APと略す）は，ファイルを単位として入出力の要求を行う．また，AP毎に利用するファイルは異なり，ファイル毎に参照頻度や入出力の傾向も異なる．さらに，複数のAPが実行されるため，同じファイルでもAP毎にファイルの参照頻度や入出力の傾向が異なる．プロセッサ処理に比べて，処理が遅い入出力処理を効率よく実行するには，これらの違いをうまく考慮したバッファキャッシュ制御法が必要である．

そこで，ここでは，APが操作したファイルの情報（以降，ファイル操作情報と略す）を収集して，APのファイル使用頻度を決定し，ファイル使用頻度に基づきバッファキャッシュを制御する手法を提案する．提案方式では，現在のファイルの参照状態，過去にファイルを利用した回数，およびファイルの大きさに着目し，ファイルの重要度を決定する．重要度が高いファ

† 岡山大学工学部
Faculty of Engineering, Okayama University
†† 岡山大学大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama University

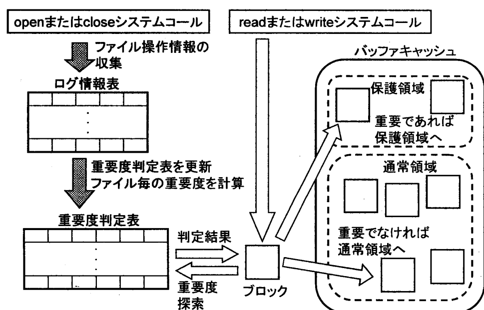

図 1　基本方式

イルのブロックを保護領域に格納し，優先して保持する．これにより，再利用する可能性の高いファイルを構成するブロックを保持し，AP の性能を向上させることを目指す．

2. ファイルの使用頻度に基づくバッファキャッシュ制御法

2.1 基本方式

基本方式を図 1 に示し，以下に説明する．提案方式では，バッファキャッシュを保護領域と通常領域に分割して管理する．ファイル操作情報として，open または close システムコールに着目し，ファイル操作情報をログ情報表に収集する．重要度の更新契機が来ると，ログ情報表を基にファイル毎に重要度を計算し，重要度判定表を更新する．read または write システムコールが発行され，ブロック読み込み要求が発生すると，重要度を基にバッファキャッシュの制御を行う．バッファキャッシュ制御の具体的な処理の流れを以下に述べる．

(1)　read または write システムコールが発行され，新規ブロック読み込み要求が発生した場合，通常領域から LRU 方式で置き換え対象のブロックを選択する．通常領域にブロックがなければ，保護領域内で最も重要度の低いファイルを構成するブロックをすべて通常領域に移し，通常領域から LRU 方式で置き換え対象のブロックを選択する．

(2)　ブロックの読み込みが終了すると，重要度を基に，読み込んだブロックのファイルが重要であるか判定し，重要であれば保護領域へ，重要でなければ通常領域へ格納する．

2.2 重要度の考え方

重要度とは，読み込んだファイルを構成するブロックを保護領域に格納するか決定するために使用する値である．重要度は，過去に open または close システムコールが発行されたときの情報を基に，将来のファイルアクセスを予測し，計算したものである．

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)　サイズの大きいファイルを保護することによって発生するバッファキャッシュの占有を防ぐため，サイズの大きいファイルほど重要度が低くなること

2.3 課　　題

提案方式では，重要度の計算方法によって性能が大きく変化する．また，重要度によるバッファキャッシュ制御でキャッシュヒット率を向上させるために，ファイル操作情報として適切な情報を収集する必要がある．さらに，ファイル操作情報を適切に重要度に反映するため，重要度判定表の更新方法を決定しなくてはならない．以上の理由により，提案方式には，以下の 3 つの課題が存在する．

(1)　重要度の計算方針

(2)　収集するファイル操作情報

(3)　重要度判定表の更新方法

以上の課題の対処を 2.4 項にて述べる．

2.4 対　　処

2.4.1 重要度の計算方針

＜重要度計算に利用する情報＞

2.2 節に述べた目標を達成するため，重要度計算にはファイル毎に以下の情報を利用する．

(1)　参照数

(2)　オープン回数

(3)　ファイルサイズ

参照数とは，現在ファイルをオープンしているプロセスの数である．参照数は，現在のオープン状態を示すため，参照数が大きいほど今後読み書きが行われる可能性が高い．

オープン回数とは，ファイルをオープンするシステムコールの発行回数である．既存のファイルシステムでは，ファイルを読み込む前に必ずファイルのオープンを行うため，ファイルの読み込み回数とオープン回数には関連性がある．また，今までにオープンされた回数の大きいファイルは，今後オープンされる可能性が高いと考えられる．

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．サイズの大きいファイルを読み込むと，単一のファイルによってバッファキャッシュが占有される可能性があり，キャッシュヒット率を低下させる原因となる．また，頻繁にオープンされるファイルのサイズは比較的小さい[9]．これらのこと

から，ファイルサイズを利用する．

<重要度の計算法>

上記の3つの情報のうち，参照数は現在ファイルがオープンされている状態を示すため，過去にオープンした履歴であるオープン回数と比べて，参照数の大きいファイルは今後読み書きされる可能性が高いと推察できる．ファイルサイズは，バッファキャッシュを制御する観点から重要度計算に利用しているため，今後ファイルを読み書きされる可能性に対して直接的に影響しない．

したがって，今後読み書きが行われる可能性を考慮し，重要度計算に与える影響を以下のようにする．

（参照数）＞（オープン回数）＞（ファイルサイズ）

2.4.2　収集するファイル操作情報

ファイル操作情報には，重要度計算に必要な情報である参照数，オープン回数，およびファイルサイズを格納する．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．また，2つのファイルの重要度が等しい場合に順位を付けるため，時刻が必要となる．提案方式では重要度計算にオープン回数を用いているため，オープン回数に関連した最新オープン時刻が必要である．

これにより，提案方式では，ファイル操作情報として，重要度計算に必要な情報，iノード番号，および最新オープン時刻を用いる．

2.4.3　重要度判定表の更新方法

<重要度判定表を更新する契機>

重要度判定表を更新する契機を判定表更新契機と呼ぶ．

ファイルのオープンまたはクローズの状況は常に変動する．ファイルのオープンまたはクローズがない状況で重要度判定表の更新が行われると，更新する情報が蓄積されていないのに更新処理が実行され，処理にかかるオーバヘッドが大きくなる．また，ファイルのオープンまたはクローズが頻繁に行われている状況で重要度判定表の更新が行われないと，ファイル操作情報を重要度判定表に適切に反映できない．また，重要度判定表の更新時の重要度判定表のファイル操作情報が多くなり，オーバヘッドが増加する．したがって，判定表更新契機は，ファイルオープンまたはクローズの操作回数に連動するのが好ましい．

判定表更新契機について，以下の3つの案が考えられる．案1〜3の特徴について，表1に示し，以下に説明する．

表1　判定表更新契機案

案	長所	短所
1	・ログ情報表を無駄なく扱える	・ファイルのオープンまたはクローズがあるのに更新が行われない可能性がある
2	・処理中に使用されるファイル数を把握する必要がない	・ファイルのオープンまたはクローズがなくても更新処理が発生する
3	・処理中に使用されるファイル数を把握する必要がない	・表に蓄えられる容量を超えた場合について別途に対策が必要である

(案1)　ログ情報表に一定数のファイル情報が貯まったとき

(案2)　前回の重要度判定表の更新から一定時間が経過したとき

(案3)　前回の重要度判定表の更新から一定回数ファイルがオープンまたはクローズされたとき

案2は，ファイルのオープンまたはクローズの操作回数に連動しないため，判定表更新契機として用いない．

案1では，一定数のファイル操作情報を蓄えられるだけのログ情報表を用意すればよく，常に表を限界まで利用できる．しかし，ログ情報表に蓄えられているファイル数が一定数に達しなければ，ファイルのオープンまたはクローズが多発していても更新が行われないという問題がある．

案3を更新契機とする場合，ログ情報表のエントリ数を把握する必要がない．しかし，ログ情報表を意識しないため，一定回数のファイルのオープンまたはクローズが行われる前に，ログ情報表があふれることがある．

そこで，案1と案3を組み合わせて採用することにより，それぞれの問題点を解決する．

<重要度判定表を更新する方法>

ファイル操作情報のうち，オープン回数は，単純に加算すると，古い情報と新しい情報が等しく扱われる．そこで，最近のオープン操作をより大きく重要度に反映させるため，古い情報ほど影響を小さくする．

これらを考慮して重要度判定表を更新する方法として，以下の2つの案が考えられる．

(案1)　過去n期間の情報を利用する手法

(案2)　重み付け手法

案1を図2に示す．案1は，最新の期間iから過去n期間に，ログ情報表に登録または更新された情報を重要度判定表の更新に使い，それよりも古い期間の情報を切り捨てる．

案2を図3に示す．案2は，重要度判定表を更新するときに，古い情報ほど影響を小さくする重み関数をかけることで，古い情報の影響を小さくする．

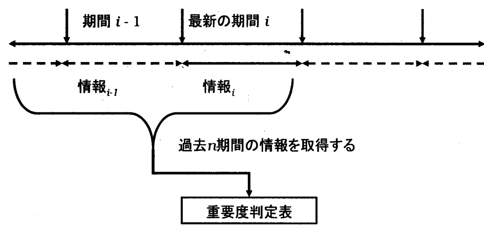

図 2　過去 n 期間の情報を利用する判定表更新法

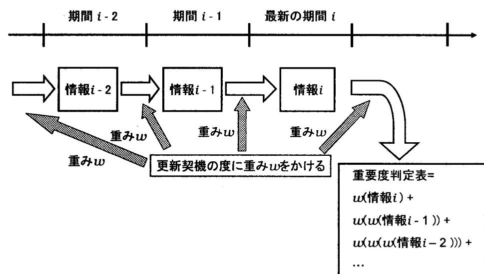

図 3　重み付けによる判定表更新法

案1は，ファイルのオープンまたはクローズの流れを期間毎に区切って保存するという特徴を持つ．この特徴により，過去の情報から将来のアクセスパターンを予測することが可能である．しかし，案1では期間別にログ情報表を扱う必要があり，判定表更新契機が来ると必ずメモリコピーを行うために，処理負荷が大きいという欠点を持つ．

これに対し，案2は処理自体が単純であり，重要度判定表を作成する際にメモリコピーを行う必要がない．このため，案2は，案1と比較して高速である．これより，提案方式では，案2を採用し，この処理法を判定表更新処理法と呼ぶ．

2.5　期待される効果

(1)　AP 使用時の入出力性能の向上

AP の入出力振る舞いに合わせたバッファキャッシュ制御により，AP 使用時の入出力性能を向上させる．

(2)　サイズの大きいファイル利用時の運用中のサービスへの影響を抑制

LRU 方式では，サービス運用中にサイズの大きいファイルを1度オープンして読み込むと，このファイルを構成するブロックによってバッファキャッシュが埋め尽くされ，運用中のサービスで利用されているブロックがバッファキャッシュから解放されてしまう．提案方式では，ファイルサイズを意識することにより，サイズの大きいファイルによってバッファキャッシュが埋め尽くされることを防ぎ，運用中のサービスの入出力性能が低下することを抑制する．

(3)　バックアップによる運用中のサービスへの影響を抑制

バックアップ処理では，基本的にファイルを1度しかオープンしない．また，バックアップ処理は，入出力負荷が大きいという特徴を持っている．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，バッファキャッシュがバックアップ処理で使用されたブロックで埋め尽くされ，運用中のサービスで利用されているブロックがバッファキャッシュから解放されてしまう．提案方式では，1度しかオープンされないファイルは重要度が低いため，バックアップ処理で利用されるブロックは，保護されにくいという特徴を持っている．これにより，運用中のサービスで利用されているブロックが保護され，運用中のサービスの入出力性能が低下することを抑制する．

3.　実 現 方 式

3.1　処理の流れ

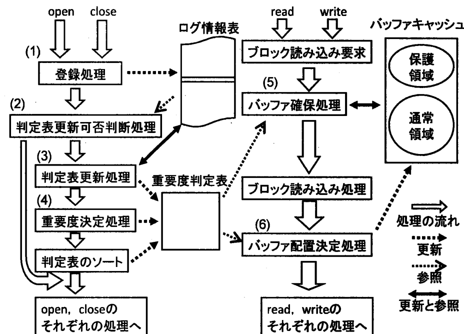

図 4　処理の流れ

提案方式の処理の流れを図4に示す．提案方式の処理の契機には，open，close，read，および write のシステムコールが存在する．open と close のシステムコールは，ファイル操作情報を収集するための契機である．read と write のシステムコールは，バッファキャッシュ制御を行うための契機である．

open または close システムコールが発行されると，(1) 登録処理によって，ログ情報表にファイル操作情報を収集する．次に，(2) 判定表更新可否判断処理を行い，判定表の更新条件を満たさない場合は，open または close システムコールのそれぞれの処理に戻る．判定表の更新条件を満たす場合には，(3) 判定表更新処理によって，重要度判定表を更新する．判定表の更新が終了したら，(4) 重要度決定処理によって重要度を

表 2　ログ情報表での各情報の登録，更新契機

情報	登録，更新契機
参照数	ファイルのオープンまたはクローズ時
オープン回数	ファイルのオープン時
ファイルサイズ	ファイルのオープンまたはクローズ時
i ノード番号	新規エントリ作成時
最新オープン時刻	ファイルのオープン時

計算し，重要度判定表を重要度順にソートして，open または close システムコールのそれぞれの処理に戻る．

read または write システムコールが発行され，ブロック読み込み要求が発生したときに，(5) バッファ確保処理によって，通常領域から LRU で置き換え対象のブロックを選択し，バッファの確保を行う．通常領域にブロックがなければ，保護領域内で最も重要度の低いファイルを構成するブロックをすべて通常領域に移し，通常領域から LRU で置き換え対象のブロックを選択して，バッファの確保を行う．ブロック読み込み処理が終了すると，(6) バッファ配置決定処理によって，読み込んだブロックの構成するファイルの重要度が高ければ保護領域に，重要度が低けれれば通常領域にそれぞれ格納し，read または write システムコールのそれぞれの処理に戻る．

3.2　各処理の設計

3.2.1　登 録 処 理

登録処理では，ファイル操作情報を収集し，ログ情報表に登録する．当該ファイルの操作情報が既にログ情報表に登録されている場合は，ファイル操作情報を更新する．ログ情報表のファイル操作情報と，登録処理内での各情報の更新契機を表 2 に示す．

参照数は，現在ファイルをオープンしている状態を示すため，ファイルがオープンまたはクローズされたときに更新する．オープン回数は，ファイルがオープンされたときに更新する．ファイルサイズは，現在のファイルのサイズを示すため，ファイルがオープンまたはクローズされたときに更新する．i ノード番号はファイルが作られてから変化しないため，新規にエントリを登録するときに登録し，以降更新は行わない．最新オープン時刻は，ファイルがオープンされたときに更新する．

3.2.2　判定表更新可否判断処理

判定表更新可否判断処理では，2.4.3 項の重要度判定表の更新契機で述べた判定表の更新条件に基づき，重要度判定表を更新するか否かの判断を行う．このときに判定表更新契機であれば，判定表更新処理に移行し，判定表更新契機でなければ open または close システムコールのそれぞれの処理に戻る．

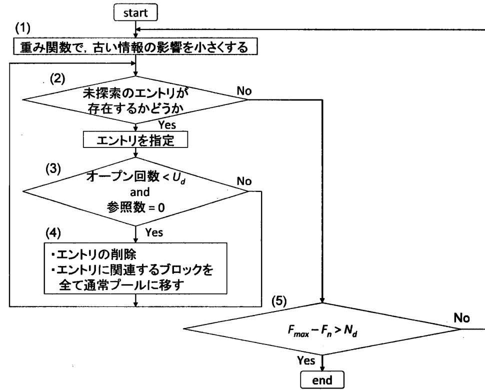

図 5　判定表更新処理

(1) 参照数	(2) ファイルサイズ	(3) オープン回数	(4) iノード番号	(5) 最新オープン時刻	(6) 重要度
		⋮			

図 6　重要度判定表

3.2.3　判定表更新処理

判定表更新処理では，2.4.3 項で述べた判定表更新処理法に基づき，重要度判定表の更新を行う．判定表更新処理の流れを図 5 に，重要度判定表を図 6 に示す．

図 5 において，F_n は重要度判定表に現在蓄えられているエントリの数，F_{max} は重要度判定の対象となるファイル数の最大値である．判定表更新処理では，(1) 重み関数で古い情報の影響を小さくする．次に，(2) エントリを探索し，(3) 探索したエントリのオープン回数が閾値 U_d 以下でかつ参照数が 0 であるか判定する．(3) の条件を満たさない場合には，(2) 次のエントリを探索する．(3) の条件を満たす場合，(4) このエントリを重要度判定表から削除し，エントリに関連するブロックすべてを保護領域から通常領域に移す．これは，重要度判定表から削除されたエントリに関連するブロックを保護することによって，キャッシュヒット率が低下することを防ぐためである．すべてのエントリの探索が終了すると，(5) 重要度判定表の空きエントリ数が閾値 N_d を超えているかどうか判定する．閾値 N_d を超えていれば，判定表更新処理を終了し，超えていなければ，重要度判定表の空きエントリ数が閾値 N_d 個以上のエントリを超えるまで，繰り返し (1)～(4) の操作を行う．

図 6 において，重要度判定表のエントリには，ファイル操作情報に加え，バッファキャッシュ制御を行うための重要度が必要である．よって，3.2.1 項で述べ

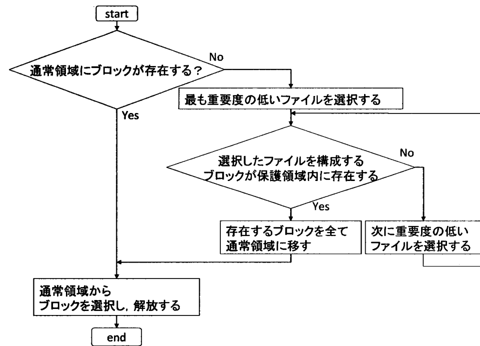

図 7 バッファ確保処理

たログ情報表のエントリの末尾に (6) 重要度を追加したものを重要度判定表のエントリとする．

3.2.4 重要度決定処理

重要度決定処理では，2.4.1 項で述べた重要度の計算方針に基づき，重要度判定表に含まれるすべてのエントリに対して重要度を計算する．

3.2.5 バッファ確保処理

バッファ確保処理の流れを図 7 に示し，以下に説明する．バッファ確保処理では，新規にブロック読み込み要求が発生したときに，バッファキャッシュからブロックを解放して，新しいブロックを読み込むための領域を確保する．このとき，通常領域にブロックがあれば，通常領域からブロックを解放する．通常領域にブロックがなければ，重要度判定表から最も重要度の低いファイルを指定する．このファイルを構成するブロックが保護領域内に存在すれば，これらすべてを通常領域へ移し，このうち 1 つを解放する．このファイルを構成するブロックが 1 つもない場合は，重要度判定表から，次に重要度の低いファイルを選択し，上記の操作を繰り返す．

3.2.6 バッファ配置決定処理

バッファ配置決定処理は，重要度に基づき，読み込みが終了したブロックを保護領域に配置するか，通常領域に配置するか決定する処理である．読み込んだブロックの構成するファイルが，重要であると判断されれば保護領域に，重要でないと判断されれば通常領域にそれぞれ繋ぐ．このとき，ブロックの構成するファイルが重要であるか否かを判断する基準が必要となる．重要であるか否か判定する基準を重要度判定法と呼ぶ．

重要度判定法の案として，以下の 2 つが挙げられる．案 1，案 2 の特徴を表 3 に示す．

(案 1) 重要度に対して下限 I_{min} を設定し，重要度が I_{min} よりも高ければ重要であるとする方式

表 3 各重要度判定法の特徴

案	長所	短所
1	無駄なブロックの保護が発生しない	重要度の変化に対応できない
2	重要度の変化を意識しなくてよい	無駄なブロックの保護が発生する

(案 2) 重要度判定表の上位から F_v 個のファイルを重要であるとする方式

案 1 は，重要度に対して最低値を設定できるため，I_{min} を適切に設定することができれば，常に重要なファイルを構成するブロックだけを保護でき，無駄なブロックの保護が発生しない．しかし，重要度計算に利用する参照数，オープン回数，およびファイルサイズは処理中に複雑に変動し，重要度も複雑に変化する．このため，案 1 では，重要度の変化に十分に対応することができない．

案 2 は，不要と考えられているブロックまで保護する可能性があるため，無駄なブロックの保護が発生する可能性がある．しかし，重要度判定表に対して重要なファイル数を設定するため，重要度の変化を意識する必要がない．このため，処理に対しての適応性が案 1 よりも優れている．

処理に合わせた重要度判定が行われなければ，提案方式の汎用性を失ってしまうことになる．よって，重要度判定法は案 2 を採用する．

ここで，バッファ配置決定処理によって重要から非重要になったファイルを構成するブロックの扱いについて述べる．バッファ配置決定処理では，現在保護領域に格納されている非重要ファイルを構成するブロックを通常領域に移さない．この理由を次に示す．

(1) 判定表更新処理において，オープン回数が少なく，参照数が 0 であるファイルのエントリは，重要度判定表から削除される．一度重要であると判断されたファイルは，一度も重要であると判断されたことのないファイルよりも，次回の判定表更新処理によって，再び重要であると判断される順位に上がる可能性が高く，再び読み込まれる可能性が高いといえる．これにより，非重要になったファイルを構成するブロックを引き続き保護しておくことで，キャッシュヒット率の向上が予測される．

(2) 一度重要であると判断されたファイルは，一度も重要であると判断されたことのないファイルに比べ，比較的サイズの小さいファイルであると推測される．よって，このファイルを保護することによってバッファキャッシュの占有が発生する可能性は低い．

(3) バッファ配置決定処理によって重要から非重要

になったファイルを通常領域に移すには，保護領域全体から非重要になったファイルを構成するブロックを検索し，ブロックを移す必要がある．バッファ配置決定処理の度にこれらの処理を行うと，read または write システムコールを行うときのオーバヘッドが大きくなり，提案方式の性能低下につながる．

以上の理由より，バッファ配置決定処理では，現在保護されているブロックを構成するファイルが非重要になったときに，非重要ファイルを構成するブロックを保護領域から通常領域に移さない．

4. 評　　価

4.1 評 価 環 境

提案方式を FreeBSD 4.3-RELEASE に実装した．カーネルの make 処理の実行時間を測定し，従来方式（LRU）と比較した．評価の環境を以下に示す．

(1) CPU : Pentium4（1.95GHz）

(2) メモリ : 512MB

(3) バッファキャッシュの大きさ : 3.0MB

なお，提案方式の効果を把握しやすくするため，バッファキャッシュ内に該当するブロックが見つからなかった場合に，実メモリ上に該当するページがあるかどうかを探索する機能（VMIO 機能）を無効にして測定した．

4.2 評価において決定する計算式と閾値

2.4.1 項で述べた重要度の計算方針に基づき，重要度の計算式を式 (1) とした．式 (1) において，I は重要度，N_{open} はオープン回数，F_{num} は参照数，k は定数，F_{size} はファイルサイズ，B_{size} は 1 ブロックのサイズである．

$$I = \frac{N_{open} + F_{num} \times k}{[\frac{F_{size}}{B_{size}}] + 1} \tag{1}$$

なお，式 (1) では，オープン回数と参照数の重要度計算への影響の違いを考慮し，定数 k を導入した．本評価では，$k = 10$ とした．

2.4.3 項で述べた重み関数 w を式 (2) とした．式 (2) において，N_{open} はオープン回数，Dec は重みである．

$$w(N_{open}) = N_{open} \times Dec \tag{2}$$

評価における各閾値を以下に示す．

(1) R_c : 重要度を計算した直後から open または close システムコールが発行された回数の総和が R_c になると，重要度を再計算（重要度再計算契機）

(2) F_v : 重要度に基づき重要ファイルと判定するファイルの数（重要度ファイル数）

(3) F_{max} : 重要度判定の対象となるファイル数の最大値

(4) N_d : 登録するファイル数が F_{max} になると，N_d 個のファイルを重要度判定表から削除

(5) U_d : 判定表更新処理において，オープン回数が U_d 以下のものをまとめて重要度判定表から削除

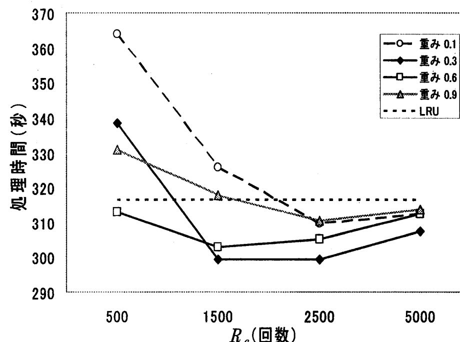

図 8　重要度再計算契機が処理時間に与える影響

4.3 評 価 結 果

<重要度再計算契機が処理時間に与える影響>

重要度再計算契機（R_c）が処理時間に与える影響を，図 8 に示す．なお，F_v を 128 個，F_{max} を 512 個，N_d を 1，U_d を 0 とした．図 8 より，以下のことがわかる．

(1) R_c は，いづれの重みにおいても，処理時間を最短にする値を持つ．これは次の理由による．R_c が小さすぎる場合には，一度の更新契機間に収集できるファイル操作情報の量が少ないために，重要度の精度が低下する．また，R_c が大きすぎる場合には，重要度の更新があまり行われず，ファイルのオープンとクローズの状況に適切に対応できない．

(2) 重みは，処理時間を最短にする値を持つ．これは，次の理由による．重みが大きいと，古い情報の影響が大きくなり，新しい情報が重要であると判断されにくくなる．また，重みが小さいと，重要度判定表には新しい情報だけが残り，ファイルのオープンとクローズの長期的な予測ができなくなる．

<重要度ファイル数が処理時間に与える影響>

重要度ファイル数（F_v）が処理時間に与える影響を，図 9 に示す．なお，F_{max} は 512 個，R_c は図 8 の実験結果において重み 0.3 のときに最も処理時間を短縮した 2500 とした．また，N_d を 1 とし，U_d を 0 とし

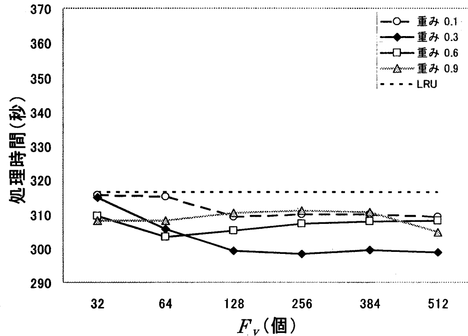

図 9 重要度ファイル数が処理時間に与える影響

た．図9より，以下のことがわかる．

(1) F_v が 128 個以上になると，処理時間はほとんど変わらない．これは，F_v が 128 個以上の場合，バッファキャッシュに読み込まれるブロックがすべて保護されることが原因である．

(2) 重みが 0.3 で，F_v が 256 個の場合，処理時間を最大 18 秒（5.7%）短縮している．このことから，提案方式は LRU に比べ，性能が向上するとわかる．

5. おわりに

AP の操作したファイルの情報を収集し，ファイルの使用頻度に基づき，優先して保持するブロックを決定し制御するバッファキャッシュ制御法を提案した．提案方式では，バッファキャッシュを保護領域と通常領域という二つの領域に分割して管理する．open と close のシステムコールに着目し，ファイル操作情報を収集する．収集した情報を基にファイル毎の重要度を算出し，重要度判定表を更新する．ここで，重要度は，参照数，オープン回数，およびファイルサイズに依存する．read と write のシステムコールを契機としたブロック読み込み要求が発生すると，重要度を基にバッファキャッシュを制御する．

提案方式を FreeBSD 4.3-RELEASE に実装し，カーネルの make 処理について評価した．この結果，提案方式は，従来方式（LRU）に比べ，処理時間を最大 18 秒（5.7 %）短縮できることを明らかにした．

残された課題として，様々な AP での評価がある．

謝辞 本研究の一部は，科学研究費補助金 若手研究 (B)（課題番号 18700030），および科学研究費補助金 基盤研究 (B)(課題番号：18300010) による．

参 考 文 献

1) J.T. Robinson and M.V. Devarakonda, "Data Cache Management Using Frequenfcy-Based Replacement," Proc. the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp.134-142, 1990.
2) E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-k Page Replacement Algorithm for Database Disk Buffering," Proc. the 1993 ACM SIGMOD Conference, pp.297–306, 1993.
3) V. Phalke and B. Gopinath, "An Inter-Reference Gap Model for Temporal Locality in Program Behavior," Proc. the USENIX Summer 1994 Technical Conference, pp.291–300, 1995.
4) P. Cao, E.W. Falten and K. Li, "Application-Controlled File Caching Policies," Proc. the USENIX Summer 1994 Technical Conference, pp.171–182, 1994.
5) J.M. Kim, J. Choi, J. Kim and S.H. Noh, "A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References," Proc. 4th Symposium on Operating System Design and Implementation (OSDI 2000), pp.119–134, 2000.
6) T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," Proc. the 20th International Conference on Very Large Databases, pp.297–306, 1993.
7) G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," Proc. the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp.115–126, 1997.
8) Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," Proc. the 1999 ACM SIGMETRICS Conference on Mesurement and Modeling of Computer Systems, pp.122–133, 1999.
9) J.K. Ousterhout, H.D. Costa, D. Harrison, J.A. Kunze, M. Kupfer and J.G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," Proc. the 10th Symposium on Operating System Principles, pp.15–24, 1985.