Vol.2014-OS-130 No.5
2014/7/28

未参照バッファ数に着目した入出力バッファ分割法

山本光一[1]　土谷彰義[1]　山内利宏[1]　谷口秀夫[1]

概要：利用者が優先して実行したい処理（優先処理）の実行処理時間を短縮する方式として，ディレクトリ優先方式を提案した．この方式は，入出力バッファを二つの領域に分割し，指定したディレクトリ直下のファイル（優先ファイル）を優先的にキャッシュする．これにより，優先処理が頻繁にアクセスするファイルを優先ファイルとすることで，優先処理の実行処理時間を短縮できる．しかし，ディレクトリ優先方式は，優先ファイル以外のファイル（非優先ファイル）のキャッシュヒット率を著しく低下させ，悪影響を生じさせることがある．そこで，本稿では，領域のサイズを更新するまでの期間内に参照されてないバッファ数に着目し，このバッファがない方の領域のサイズを増加させる方式を提案する．提案方式は，優先ファイルをキャッシュする領域の下限を設定するパラメータを用いることで，優先ファイルをキャッシュする領域のキャッシュヒット率が低下しすぎないようにしている．また，カーネル make 処理と Web サーバ処理において提案方式を評価した結果を報告する．

1. はじめに

計算機で実行される処理には，利用者が優先したい処理（以降，優先処理と略す）とそうでない処理（以降，非優先処理と略す）がある．優先処理のファイルアクセスを高速化するには，入出力バッファのキャッシュヒット率を向上させることが有効である．優先処理のキャッシュヒット率を向上させる方法として，ディレクトリ優先方式 [1] を提案した．この方式は，入出力バッファを二つの領域に分割し，指定したディレクトリ（以降，優先ディレクトリと略す）直下のファイル（以降，優先ファイルと略す）を優先的にキャッシュする．これにより，他のファイル（以降，非優先ファイルと略す）のキャッシュが優先ファイルのキャッシュを無効化することを防いでいる．つまり，優先処理が頻繁にアクセスするファイルを多く含むディレクトリを優先ディレクトリとすることにより，優先処理のキャッシュヒット率が向上し，優先処理の実行処理時間を短縮できる．

文献 [1] のディレクトリ優先方式（以降，基本方式と略す）は，優先ファイルをキャッシュする領域のサイズを単調増加させるため，非優先ファイルをキャッシュする領域のサイズは単調減少してしまう．このため，非優先ファイルのキャッシュヒット率が低下し，アクセスするファイルデータ量の総和よりも入出力バッファサイズが小さい場合，二つの問題点が発生する．一つ目の問題点は，優先処理の実行処理時間の増加である．優先処理を実行するとき，優先処理がアクセスするファイルを格納するディレクトリすべてを優先ディレクトリとした場合，ディレクトリ優先方式は有効に働かない．このため，優先処理が頻繁にアクセスするファイルを直下に多く含むディレクトリのみを優先ディレクトリに指定する必要がある．しかし，この場合，優先処理がアクセスするファイルの一部は，非優先ファイルとなる．基本方式では，非優先ファイルをキャッシュする領域のサイズが単調減少するため，優先処理がアクセスする非優先ファイルのキャッシュヒット率が低下し，優先処理全体としての実行処理時間が増加する可能性がある．二つ目の問題点は，共存する非優先処理の実行処理時間の増加である．非優先処理は，非優先ファイルにアクセスするため，非優先ファイルのキャッシュヒット率が低下し，時には 0 になる．このため，非優先処理の実行処理時間が非常に増加してしまう．

そこで，本稿では，上記の問題に対処するため，ディレクトリ優先方式において，各領域の領域のサイズを増減させる改良方式を提案する．提案方式は，各領域の領域のサイズを更新するまでの期間内に参照されてないバッファ数に着目する方式である．提案方式は，優先ファイルをキャッシュする領域の下限を設定するパラメータを用いることで，優先ファイルをキャッシュする領域のキャッシュヒット率が低下しすぎないようにしている．また，カーネル make 処理と Web サーバ処理において提案方式を評価した結果を報告する．

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

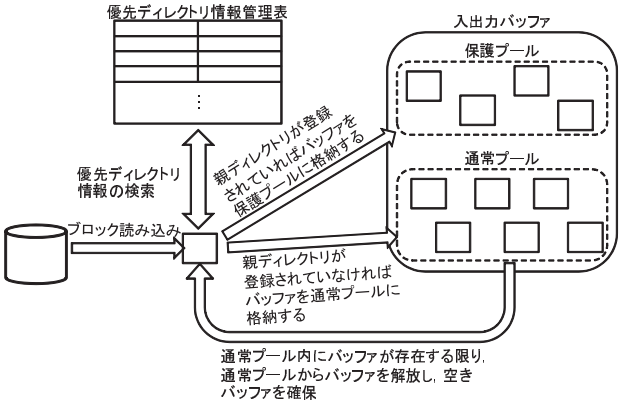

図 1 基本方式

2. ディレクトリ優先方式

2.1 考え方

文献 [1] で提案した基本方式を**図 1** に示す．ディレクトリ優先方式は，入出力バッファを保護プールと通常プールに分割し，各プール内のバッファを LRU 方式で管理する．保護プールには優先ファイルのブロックを保持するバッファを格納し，通常プールには非優先ファイルのブロックを保持するバッファを格納する．ここで，バッファとは 1 ブロックをキャッシュするためのメモリ領域を指す．入出力バッファは，このバッファを複数持ち，リスト構造で管理している．また，入出力バッファサイズは固定である．このため，一方のプールが大きくなるとそのプールの拡大分だけ他方のプールは小さくなる．

基本方式では，ブロックアクセス時，通常プール内にバッファが存在する限り，通常プールからバッファを解放し，空きバッファを確保する．空きバッファにブロックを読み込んだ後，読み込んだブロックに対応するファイルが優先ファイルであれば保護プールに，非優先ファイルであれば通常プールにバッファを格納する．また，基本方式は，以下の要求を満足するように設計されている．

（要求 1） 優先ファイルへのアクセス時間を短縮

このため，基本方式は，優先ファイルへのブロックをできるだけ多くキャッシュするように設計されており，保護プールサイズが単調増加する．これに伴い，通常プールサイズが単調増加し，非優先ファイルのキャッシュヒット率が低下する．

2.2 問題点

アクセスするファイルデータ量の総和よりも入出力バッファサイズが小さい場合における基本方式の問題点を以下に示す．

（問題 1） 優先処理の実行処理時間の増加

優先処理を実行するとき，優先処理がアクセスするファイルを格納するディレクトリすべてを優先ディレクトリとした場合，ディレクトリ優先方式は有効に働かない．このため，優先処理が頻繁にアクセスするファイルを直下に多く含むディレクトリのみを優先ディレクトリに指定する必要がある．しかし，この場合，優先処理がアクセスするファイルの一部は，非優先ファイルとなる．基本方式では，非優先ファイルをキャッシュする領域のサイズが単調減少するため，優先処理がアクセスする非優先ファイルのキャッシュヒット率が低下し，優先処理全体としての実行処理時間が増加する可能性がある．

（問題 2） 共存する非優先処理の実行処理時間の増加

非優先処理は，非優先ファイルにアクセスするため，非優先ファイルのキャッシュヒット率が低下し，時には 0 になる．このため，非優先処理の実行処理時間が非常に増加してしまう．

3. 未参照バッファ数に着目した入出力バッファ分割法

3.1 目的と考え方

（要求 1）を満たし，かつ（問題 1）を解決し，その上で可能な範囲で（問題 2）を解決する新たな入出力バッファ分割法を提案する．提案方式は，目標とする保護プールサイズ（S_{target}）を増減させることで，保護プールのキャッシュヒット率を向上させつつ，通常プールのキャッシュヒット率の低下の抑制を目的とする．提案方式は，最近 S_{target} が更新されてから参照されてないバッファ（以降，未参照バッファと略す）数に着目する．S_{target} の更新は，どちらか片方のプール内の未参照バッファがなくなり，かつキャッシュミスが発生したときに行われるため，S_{target} の更新期間は，S_{target} 更新のたびに変化する．未参照バッファがないプールは，そのプール内のバッファが頻繁に参照された，またはアクセスパターンの変化によるキャッシュミスの多発でプール内のバッファが置き換えられたことを示す．このため，そのプールのサイズを増加させることで，そのプールのキャッシュヒット率の向上が期待できる．また，保護プールサイズの下限を設定するパラメータを用いることで，保護プールのキャッシュヒット率が低下しすぎないようにする．

S_{target} に基づく制御の方針として，以下の二つがある．

（方針 1） S_{target} を動的に更新

保護プールと通常プールの各プール内の未参照バッファ数に着目し，未参照バッファがなくなったときに，S_{target} を更新する．

（方針 2） キャッシュしているブロックの有効利用

S_{target} を更新したとき，現在の保護プールサイズ（S_{cur}）と S_{target} が一致しなくなる．直ちに S_{cur} と S_{target} を一致させるため，バッファを解放した場合，解放後に解放したバッファにアクセスがあったときにキャッシュヒット率が低下する．このため，バッファを直ちに解放せず，バッ

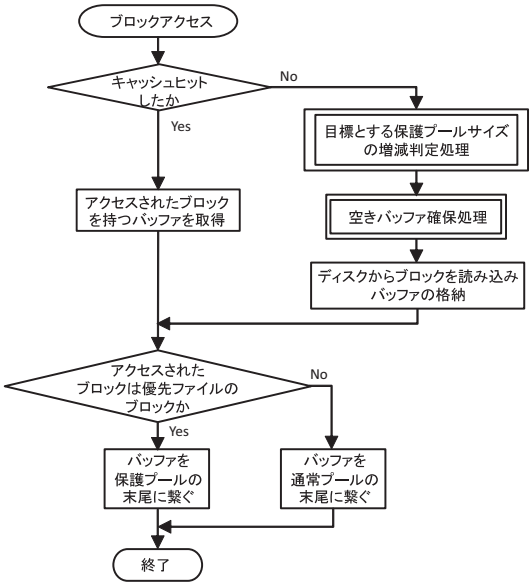

図 2 提案方式の処理の流れ

ファの解放と空きバッファの確保時に徐々に差を縮め，一致させる．

3.2 基本的な処理の流れ

提案方式の処理の流れを**図 2** に示し，以下で説明する．なお，図 2 のうち，2 重の四角で示す処理が，提案方式で新たに追加，もしくは変更する処理である．

(1) キャッシュヒットした場合，アクセスされたブロックを持つバッファを取得する．一方，キャッシュミスした場合，目標とする保護プールサイズの増減判定処理（3.4.3 項で後述する）と空きバッファ確保処理を実行する．空きバッファ確保処理は，3.3 節に示すバッファ解放規則に基づき，空きバッファを確保する処理である．空きバッファ確保処理実行後，ディスクからブロックを読み込む．

(2) 次に，アクセスされたブロックが優先ファイルのブロックであれば，バッファを保護プールの末尾に繋ぎ，非優先ファイルのブロックであれば，バッファを通常プールの末尾に繋ぐ．

3.3 バッファ解放規則

提案方式は，以下に示すバッファ解放規則に基づき，S_{cur} を更新する．

(1) 以下の規則に従い，バッファを解放するプールを選択する．

(a) $S_{cur} < S_{target}$ の場合

S_{cur} を大きくし，S_{target} に近づけなければならないため，通常プールを選択する．

(b) $S_{cur} = S_{target}$ の場合

S_{cur} を変化させないため，読み込むブロックが優先ファイルのブロックであるか否かに基づき，バッファを解放するプールを選択する．優先ファイルのブロックであれば保護プールを，非優先ファイルのブロックであれば通常プールを選択する．

(c) $S_{cur} > S_{target}$ の場合

S_{cur} を小さくし，S_{target} に近づけなければならないため，保護プールを選択する．

(2) (1) で選択したプール内に解放できるバッファが存在するか判定する．バッファが存在すれば選択したプールから，バッファが存在しなければ選択しなかったプールから LRU 方式で解放するバッファを決定する．

3.4 課題と対処

3.4.1 課題

提案方式を実現させるための課題として以下の三つがある．

（課題 1） 未参照バッファの判別法

提案方式は，各プール内の未参照バッファ数に着目し，保護プールを増減させるか否か決定する．このため，各プールの未参照バッファを判別する方法が課題となる．

（課題 2） 目標とする保護プールサイズの増減判定処理

（方針 1）より，各プール内に未参照バッファがあるか否か判定する処理が課題となる．

（課題 3） 目標とする保護プールサイズの増減量の決定法

（方針 1）より，S_{target} をどれだけ増減させるかが課題となる．

次項から上記三つの課題の対処法を示す．

3.4.2 未参照バッファの判別法

各プール内のバッファは LRU 方式で管理されているため，参照が古い順にキューに繋がれる．つまり，参照された，またはキャッシュヒットしたバッファは，自身の属するキューの末尾に繋ぎ変えられる．この特性を活かし，各プール内の未参照バッファを判別するために，保護プールと通常プールのキューに未参照バッファとそうでないバッファを判別する目印となるバッファ（以降，目印バッファと略す）を挿入する．目印バッファは，参照されず，かつキャッシュヒットしないバッファであるため，目印バッファの前に存在するバッファを参照されてないバッファとして判別できる．未参照バッファは，最近 S_{target} が更新されてから参照されてないバッファである．そこで，各プール内の未参照バッファ判別のため，目印バッファを，S_{target} の更新ごとにキューの末尾に繋ぎ変える．目印バッファを挿入した後，一定回数のブロックアクセスがあった場合の入出力バッファ内の例を**図 3** に示す．図 3 より，未参照バッファが存在する場合は，目印バッファの前に存在することが分かる．また，目印バッファがキューの先頭に達した場合，そのキューの未参照バッファがなくなったといえる．

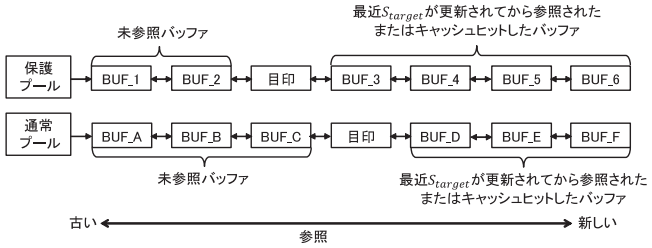

図 3 提案方式における入出力バッファ内の例

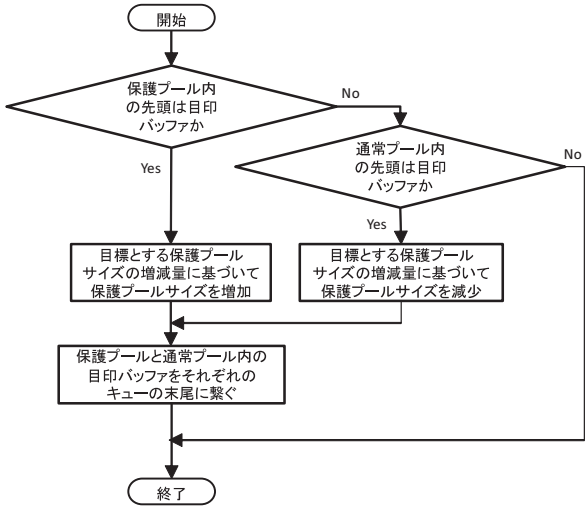

図 4 目標とする保護プールサイズの増減判定処理の流れ

3.4.3 目標とする保護プールサイズの増減判定処理

S_{target} の増減判定処理は，キャッシュミスが発生する度に行う．キャッシュミスの発生は，新たなブロックアクセスがあったことを示すため，これを契機とすることで，アクセスパターンの変化に対応する．処理の内容を**図 4** に示し，以下で説明する．

(1) 保護プール内の先頭が目印バッファの場合，目標とする保護プールサイズの増減量に基づいて S_{target} を増加させる．

(2) 通常プール内の先頭が目印バッファの場合，目標とする保護プールサイズの増減量に基づいて S_{target} を減少させる．

提案方式では，プール内の先頭が目印バッファの場合，つまりプール内に未参照バッファがない場合を S_{target} 増減の契機とする．これは，プール内に未参照バッファがない場合，プール内のバッファが頻繁に参照された，またはアクセスパターンの変化によるキャッシュミスの多発でプール内のバッファが置き換えられたことを示し，そのプールのバッファが不足していることを示すためである．このため，目印バッファが先頭のプールのサイズを増加させることで，そのプールのキャッシュヒット率の向上が期待できる．また，S_{target} の増減を行った場合，保護プールと通常プール内の目印バッファをそれぞれのキューの末尾に繋ぐ．

表 1 目標とする保護プールサイズ増減量の決定法

S_{target} の増加量の決定法	通常プールの未参照バッファのバッファ分 ただし，保護プールサイズ ≦（入出力バッファに保持できるバッファ数－先読みブロックの最大数）を維持
S_{target} の減少量の決定法	保護プールの未参照バッファのバッファ分 ただし，保護プールサイズ ≧（M ×入出力バッファに保持できるバッファ数）を維持

3.4.4 目標とする保護プールサイズ増減量の決定法

片方のプールのサイズが大きくなりすぎてしまうと，もう一方のプールのサイズが小さくなりすぎてしまうため，優先ファイルまたは非優先ファイルのキャッシュヒット率が極端に低下することを防ぐ必要がある．

そこで，各プール内の未参照バッファ数に着目して，S_{target} の増減量を決定する．決定方法を以下で述べる．

(1) 未参照バッファは，アクセスパターンの変化がない限り，次に参照される可能性が低い．このため，未参照バッファを解放した場合でも，キャッシュヒット率は低下しにくく，優先的にバッファから解放できる．

(2) 未参照バッファは優先的にバッファから解放できるため，未参照バッファの数分，S_{target} を増減できる．

S_{target} の増減量の決定法を**表 1** に示し，以下で説明する．また，提案方式は，保護プールのキャッシュヒット率が低下しすぎないようにするため，保護プールサイズの下限を設定するパラメータ M を用いる．

(a) S_{target} を増加させる場合

通常プール内の未参照バッファのバッファ分，S_{target} を増加させる．この場合，通常プールに最低限必要なバッファを確保するため，保護プールサイズの上限を（入出力バッファに保持できるバッファ数－先読みブロック数の最大数）とする．

(b) S_{target} を減少させる場合

保護プール内の未参照バッファのバッファ分，S_{target} を減少させる．この場合，保護プールサイズが（保護プールサイズの下限を設定するパラメータ M ×入出力バッファに保持できるバッファ数）を下回らないようにする．

また，この決定法は，未参照バッファのバッファ分のみ S_{target} の増減を行うため，解放されるバッファは未参照バッファである可能性が高い．S_{target} を増加させる場合の例を**図 5** に示す．図 5 の状態では，通常プール内の未参照バッファが三つ存在するため，S_{target} を 3 バッファ分増加させる．

4. 評価

4.1 評価内容

提案方式で（問題 1）と（問題 2）を解決できるか評価するため，以下の評価を行った．

(1) カーネル make 処理のみを行う評価

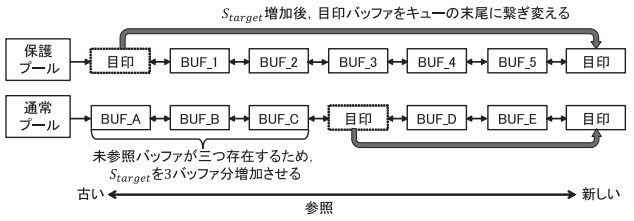

図 5 目標とする保護プールサイズ増減の例

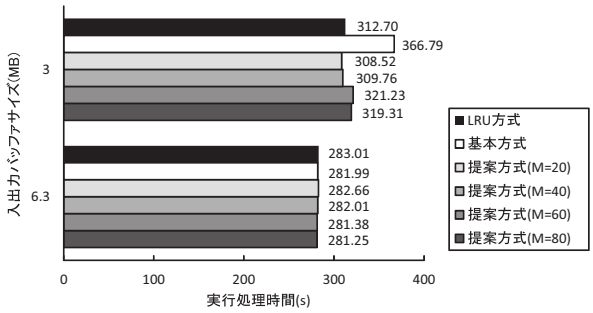

図 6 カーネル make の実行処理時間

非優先ファイルにアクセスする優先処理としてカーネル make を実行し，（問題 1）を解決できるか評価する．

(2) Web サーバ運用中に Web コンテンツのファイルをバックアップする場合の評価

Web サーバが持つトップページを構成するファイルの要求に対する Web サーバの処理を優先処理，Web サーバの他の処理と Web コンテンツのファイルをバックアップする処理を非優先処理として評価した．この評価により，（要求 1）を満たし，（問題 1）と（問題 2）を解決できるか評価する．

本評価は，LRU 方式，基本方式，および提案方式で行った．提案方式におけるパラメータ M の設定値は，M の値による影響を評価するため，20%，40%，60%，および 80% を用いた．また，本評価において，問題を解決したと判断する基準を以下のようにおき，二つの問題を解決できるか評価した．

（基準 1） LRU 方式と比べて優先処理の実行処理時間を短縮できた，もしくは同等の場合，（問題 1）を解決できたものとする．

（基準 2） 優先処理の実行処理時間を基本方式と同等，かつ基本方式と比べて，非優先処理の実行処理時間を短縮できた場合，（問題 2）を解決できたものとする．

LRU 方式は多くの場合で比較的良い性能を出せるということが知られており，一般的に多くの OS で利用されているため，LRU 方式を比較対象とした．

4.2 カーネル make 処理のみを行う評価

4.2.1 測定方法

測定前に make depend を実行した．提案方式は，make depend 実行後に S_{target} を 0 に初期化した．make depend 実行完了直後，基本方式と提案方式は優先ディレクトリに/usr/src/sys/sys/と/usr/src/sys/i386/include/を指定した．この二つのディレクトリは直下にヘッダファイルを有するディレクトリであり，直下に有するファイルデータ量の総和は約 1.6MB（338 ブロック）である．カーネル make は，この内約 1.2MB 強（254 ブロック）の優先ファイルにアクセスする．カーネル make はヘッダファイルに繰り返しアクセスし，この二つのディレクトリ直下のヘッダファイルは，他のヘッダファイルと比べ，より頻繁にアクセスされる．また，カーネル make がアクセスする非優先ファイルのファイルデータ量の総和は，約 44MB である．

4.2.2 評価環境

提案方式は，基本方式と同様に FreeBSD 4.3-RELEASE に実装した．計算機（CPU : Celeron 2.8GHz，メモリ : 768MB，OS : FreeBSD 4.3-RELEASE，VMIO : オフ，1 バッファのサイズ : 8.0KB）を用いて評価した．入出力バッファの制御方式の性能が問題となるのは，入出力バッファサイズがアクセスするファイルデータ量の総和よりも小さく，キャッシュミスが起こる場合である．このため，制御方式の違いによる性能の差を明確にするため，入出力バッファサイズを小さく制限し，3.0MB と 6.3MB の場合について測定した．ただし，システム維持のために常時確保される領域があるため，実際に利用できる領域は入出力バッファサイズより 0.7MB ほど小さい．入出力バッファサイズが 3.0MB の場合は 296 個，入出力バッファサイズが 6.3MB の場合は 720 個のバッファを保持できる．

4.2.3 評価結果

カーネル make の実行処理時間を**図 6** に示す．図 6 から以下のことがわかる．

(1) 入出力バッファサイズ 3.0MB の場合

提案方式は，M=20% で最も実行処理時間を短縮した．この場合，LRU 方式と比べて，約 4.18 秒（約 1.35%），基本方式と比べて，約 58.27 秒（約 18.89%）短縮した．また，M≤40% の場合，LRU 方式より実行処理時間を短縮できた．一方，提案方式は，M=60% で最も実行処理時間が増加した．この場合，LRU 方式と比べて，約 8.53 秒（約 2.72%）増加，基本方式と比べて，約 45.56 秒（約 14.18%）短縮した．また，M≥60% の場合， LRU 方式より実行処理時間が増加した．これは，M が増加するに伴い，通常プールを確保する領域が小さくなり，非優先ファイルのキャッシュヒット率が低下したためと考えられる．

(2) 入出力バッファサイズ 6.3MB の場合

提案方式は M=80% で最も実行処理時間を短縮した．この場合，LRU 方式と比べて，約 1.76 秒（約 0.63%），基本方式と比べて，約 0.74 秒（約 0.26%）短縮した．また，提案

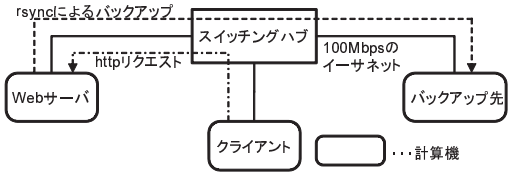

図 7 Web サーバによる評価環境

表 2 Web サーバによる評価に用いた計算機

	Web サーバ	クライアント	バックアップ先
CPU	Celeron D 2.8GHz	Celeron D 2.8GHz	Celeron 2.0GHz
メモリ	32MB	256MB	768MB
入出力バッファ	6.4MB	32MB	87MB
OS	FreeBSD 4.3-RELEASE	FreeBSD 4.3-RELEASE	FreeBSD 4.3-RELEASE
VMIO	オフ	オン	オン
1 バッファのサイズ	8.0KB	16.0KB	16.0KB

方式は，本評価におけるすべての M で，LRU 方式より実行処理時間を短縮した．

上記の結果より，入出力バッファサイズが 3.0MB の場合，M≤40% で，入出力バッファサイズが 6.3MB の場合，本評価におけるすべての M で，（問題 1）を解決できている．入出力バッファサイズが 3.0MB の場合，保護プールにアクセスされる優先ファイルをすべてキャッシュした状態のとき，通常プールサイズが小さくなりすぎてしまう．このため，M を大きくし，保護プールサイズを常に大きく保とうとした場合，非優先ファイルのキャッシュヒット率が低下し，実行処理時間が増加する．

4.3 Web サーバ運用中に Web コンテンツのファイルをバックアップする場合の評価

4.3.1 評価環境と評価方法

評価環境を**図 7** に示す．また，図 7 の各計算機の性能を**表 2** に示す．Web サーバの計算機の入出力バッファは，734 個のバッファを保持できる．

前節で述べたように，入出力バッファの制御方式の性能が問題になるのは，入出力バッファサイズがアクセスするファイルデータ量の総和よりも小さい場合である．このため，Web サーバを実行する計算機が認識できるメモリ量を 32MB に制限した．メモリを数 GB 搭載した場合であっても，Web サーバがアクセスするファイルデータ量の総和が増加すれば，提案方式は本評価と同様の効果を示すと考えられる．

Web サーバとして Apache 2.0.55 を用い，クライアントプログラムとして，ApacheBench 2.40-dev を用いた．また，バックアップ用プログラムとして rsync 2.4.6 を用いた．Apache 2.0.55 には，入出力バッファを介さずにファイルを転送する sendfile() システムコールを利用する機能がある．入出力バッファの制御方式の評価を行うため，本評価ではこの機能を無効にした．

表 3 評価で用いたファイルの情報 (100,000 回要求)

	全体	優先ファイル	6 部局のトップページを構成するファイル
ディレクトリ数（個）	1,365	14	14
ファイル数（個）	8,849	877	118
合計要求回数（回）	100,000	59,467	21,391
合計ファイルデータ量（MB）	600.0	11.1	0.7

表 4 バックアップしたファイルの情報

	全体	優先ファイル
合計ファイルデータ量 (MB)	2818.5	13.9

岡山大学の Web サーバ（www.okayama-u.ac.jp）のディレクトリ構造を再現し，2006 年 7 月の岡山大学の Web サーバへの要求から 100,000 回を抽出し，Web サーバにアクセスした．測定前に 100,000 回 Web サーバにアクセスすることにより，入出力バッファに Web コンテンツのファイルがキャッシュされている状態にした．この 100,000 回のアクセス終了後，バックアップ処理と次の 100,000 回のアクセス（アクセス間隔 100ms）をほぼ同時に開始した．また，提案方式では，最初の 100,000 回のアクセスの前に，S_{target} を 0 に初期化した．バックアップ処理は，再現した Web コンテンツのファイルをバックアップする．本稿では，バックアップ処理動作中の Web サーバの平均応答時間とバックアップ処理の実行処理時間を示す．

本評価では，文献 [1] と同様に，6 部局のトップページを構成するファイルの要求に対する Web サーバの処理を優先処理，Web サーバの他の処理と Web コンテンツのファイルをバックアップする処理を非優先処理とした．ここで，トップページを構成するファイルとは，トップページの HTML ファイルとこの HTML ファイルから参照される画像ファイルのことである．また，優先処理のキャッシュヒット率を向上させるため，岡山大学の Web サーバが持つ 6 部局のトップページを構成するファイルを直下に有するディレクトリをすべて優先ディレクトリに指定した．このため，本評価では，優先処理は優先ファイルにのみアクセスする．評価で用いたファイルの情報を**表 3** に示し，バックアップしたファイルの情報を**表 4** に示す．

4.3.2 評価結果

6 部局のトップページを構成するファイルの平均応答時間と優先ファイルの平均応答時間を**図 8** に，非優先ファイルの平均応答時間と全ファイルの平均応答時間を**図 9** に，およびバックアップ処理の実行処理時間を**図 10** に示す．

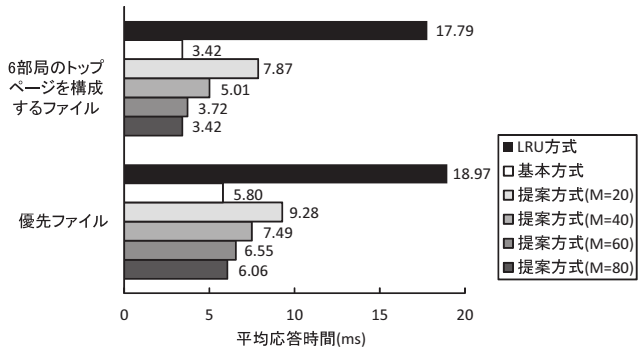

図 8 6 部局のトップページを構成するファイルの平均応答時間と優先ファイルの平均応答時間（バックアップ処理動作中）

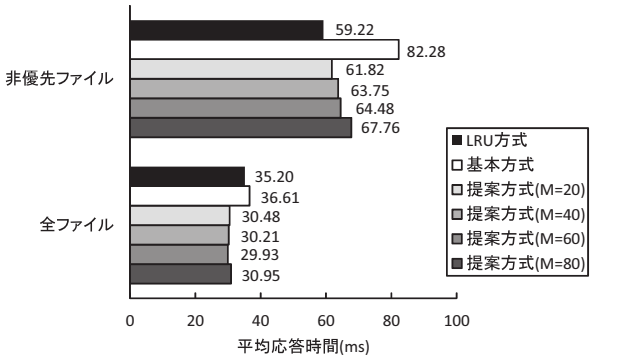

図 9 非優先ファイルの平均応答時間と全ファイルの平均応答時間（バックアップ処理動作中）

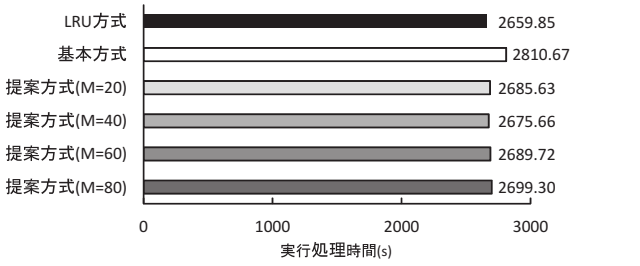

図 10 バックアップ処理の実行処理時間

図 8，図 9，および図 10 から，以下のことが分かる．

M=20% の場合，提案方式は，非優先ファイルの平均応答時間を最も短縮し，基本方式と比べ，約 20.46 ミリ秒（約 24.88%）短縮した．このときの非優先ファイルの平均応答時間は，LRU 方式と比べ，約 2.6 ミリ秒（約 4.39%）増加した．また，6 部局のトップページを構成するファイルの平均応答時間は，LRU 方式と比べ，約 9.92 ミリ秒（約 55.76%）短縮，基本方式と比べ，約 4.45 ミリ秒（約 130.11%）増加した．

M=80% の場合，提案方式は，6 部局のトップページを構成するファイルの平均応答時間を最も短縮し，LRU 方式と比べて，約 14.37 ミリ秒（約 80.79%）短縮した．このときの 6 部局のトップページを構成するファイルの平均応答時間は，基本方式と同等であった．また，非優先ファイルの平均応答時間は，LRU 方式と比べ，約 8.54 ミリ秒（約 14.4%）増加，基本方式と比べ，約 14.52 ミリ秒（約 17.65%）短縮した．

提案方式は，M の値が小さいとき，非優先ファイルの平均応答時間をより短縮できた．しかし，6 部局のトップページを構成するファイルの平均応答時間は，基本方式と比べ，大幅に増加した．一方，M の値が大きいとき，6 部局のトップページを構成するファイルの平均応答時間を基本方式と同等，かつ非優先ファイルの平均応答時間を基本方式と比べて短縮した．このため，提案方式は，M の値が大きいとき，（要求 1）を満たし，（問題 1）と（問題 2）を解決できたといえる．

5. 関連研究

入出力バッファを分割して管理する方式 [2]–[12] が提案されている．2Q[2]，LIRS[3]，および DULO[4] は入出力バッファを静的に分割し，文献 [5] の方式は重要と判断したファイルをできるだけ多くキャッシュする．一方，ARC[6,7]，CAR[8]，UBM[9]，PCC[10]，CAP[11]，および Karma[12] は，分割領域のサイズを自動的に決定する．

ARC[6,7] と CAR[8] は，入出力バッファを二つの領域に分割し，各領域から破棄されたブロックの情報を一定量保持しておき，各領域のサイズの決定に利用する．

UBM[9]，PCC[10]，および CAP[11] は，アクセスパターンを分類し，アクセスパターン毎に領域を割り当てる．Karma[12] は，ヒントとして与えられたアクセス頻度とアクセスパターンにより，ブロック群を互いに素な集合に分割し，各集合に入出力バッファの分割領域を割り当てる．UBM[9]，PCC[10]，CAP[11]，および Karma[12] は，ブロックアクセス時に，入出力バッファ全体のキャッシュヒット率が最も高くなるように，各領域のサイズをバッファ 1 つ分ずつ変更する．

文献 [2]–[12] の方式は，システム全体でのキャッシュヒット率の向上を目的とした方式であり，特定の優先処理の実行処理時間を短縮することはできない．一方，提案方式は，優先処理が頻繁にアクセスするファイルのブロックを優先的にキャッシュし，優先処理の実行処理時間を短縮できる．また，一方の領域のキャッシュヒット率を向上させ，かつ他方のキャッシュヒット率の低下を抑制するように，領域の分割サイズを決定する．

6. おわりに

入出力バッファを分割し，分割した各領域の領域サイズを更新するまでの期間内に参照されてないバッファ（未参照バッファ）数に着目し，各領域の分割サイズを増減させる式を提案した．未参照バッファがない領域は，領域内のバッファに頻繁にアクセスがあった，またはアクセスパ

ターンの変化によるキャッシュミスの多発で領域内のバッファが置き換えられたことを示す．そこで，提案方式は，分割した各領域内に未参照バッファがない場合，それに該当する領域のサイズを増加させる．

提案方式をカーネル make 処理の評価，および Web サーバ処理とバックアップ処理の共存実行処理で評価した結果を報告した．カーネル make 処理の評価の場合，基本方式が LRU 方式よりも実行処理時間が非常に増加しているとき，提案方式は，基本方式より実行処理時間を短縮し，設定するパラメータの値次第では，LRU 方式よりも実行処理時間を短縮できることを示した．提案方式は，実行処理時間を最も短縮できたパラメータにおいて，LRU 方式と比べ，約 4.18 秒（約 1.35%），基本方式と比べ，約 58.27 秒（約 18.89%）実行処理時間を短縮した．

また，Web サーバ処理とバックアップ処理の共存実行処理の評価の場合，提案方式は，優先処理がアクセスするファイルの平均応答時間を LRU 方式より短縮し，かつ非優先ファイルの平均応答時間を基本方式よりも短縮した．提案方式は，優先処理がアクセスするファイルの平均応答時間を最も短縮できたパラメータにおいて，LRU 方式と比べ，優先処理がアクセスするファイルの平均応答時間を約 14.37 ミリ秒（約 80.79%）短縮した．また，基本方式と比べ，非優先処理がアクセスするファイルの平均応答時間を約 14.52 ミリ秒（約 17.65%）短縮した．このとき，優先処理がアクセスするファイルの平均応答時間は基本方式と同等であり，優先処理がアクセスするファイルの平均応答時間を増加させることなく非優先ファイルの平均応答時間を短縮した．

謝辞 本研究の一部は，科学研究費補助金基盤研究（B）（課題番号：24300008）による．

参考文献

[1] 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 D，Vol. J91-D，No. 2，pp.435–448 (2008).

[2] Johnson, T. and Shasha, D.: 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm, *Proc. the 20th International Conference on Very Large Databases*, pp. 439–450 (1994).

[3] Jiang, S. and Zhang, X.: LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance, *Proc. the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 31–42 (2002).

[4] Ding, X., Jiang, S. and Chen, F.: A Buffer Cache Management Scheme Exploiting Both Temporal and Spatial Localities, *ACM Transactions on Storage*, Vol. 3, No. 2 (2007).

[5] 片上達也，田端利宏，谷口秀夫：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，情報処理学会論文誌：コンピューティングシステム（ACS），Vol. 3，No. 1，pp. 50–60 (2010).

[6] Megiddo, N. and Modha, D. S.: ARC: A SELF-TUNING, LOWOVERHEAD REPLACEMENT CACHE, *Proc. the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pp. 115–130 (2003).

[7] Kim, Y. J. and Anggorosesar, A.: Device-Aware Cache Management based on Adaptive Replacement, *Proc. the 9th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS '10)*, pp. 85–89 (2010).

[8] Bansal, S. and Modha, D. S.: CAR: Clock with Adaptive Replacement, *Proc. the 3rd USENIX Conference on File and Storage Technologies (FAST '04)*, pp. 187–200 (2004).

[9] Kim, J. M., Choi, J., Kim, J., Noh, S. H., Min, S. L., Cho, Y. and Kim, C. S.: A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References, *Proc. the 4th Symposium on Operating System Design and Implementation (OSDI 2000)*, pp. 119–134 (2000).

[10] Gniady, C., Butt, A. R. and Hu, Y. C.: Program-Counter-Based Pattern Classification in Buffer Caching, *Proc. the 6th Symposium on Opearting Systems Design and Implementation (OSDI 2004)*, pp. 395–408 (2004).

[11] Gupta, R. and Shrawankar, U.: Cache Access Pattern Based Algorithm for Performance Improvement of Cache Memory Management, *WSEAS Transactions on Information Science and Applications*, Vol. 10, pp. 271–284 (2013).

[12] Yadgar, G., Factor, M. and Schuster, A.: Karma: Know-it-All Replacement for a Multilevel cAche, *Proc. the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, pp. 169–184 (2007).