# 急激なメモリ需要の変動に追従するメモリ管理機構の設計と評価

永松良一 河合栄治 砂原秀樹

奈良先端科学技術大学院大学 情報科学研究科

**概要**

メモリの大容量化及び低価格化により，ディスクアクセスを抑え，サービススループットを向上させるため，多くのデータをメモリ上に格納することができるようになった．しかし，仮想マシン環境のように急激にメモリ需要が変動する環境ではメモリ不足時にページアウトが発生し，結果としてサービススループットを損なってしまうことがある．そこで本研究ではページアウト・ページインによらないメモリ管理機構を提案する．これにより，メモリ需要の変動に対して性能劣化の抑制を図る．

# Design and Evaluation of a Memory Management Mechanism Concidering Drastic Changes in Memory Demand

Ryoichi Nagamatsu Eiji Kawai Hideki Sunahara

Graduate School of Information Science, Nara Institute of Science and Technology

**Abstract**

Increasing capacity and decreasing price of main memory make more and more data stored in it to improve service throughput by suppressing disk accesses. However, such huge data in main memory can degrade the service performance when a drastic change in memory demand occurs, which causes numerous page-outs. A typical example in which this phenomenon can occur is a virtual machine environment. In this study, we propose a memory management mechanism that does not have resort to the page-out/page-in mechanism. This mitigates the severe performance penalty caused by page-outs.

## 1 研究背景

近年では，web サーバやデータベースのメモリキャッシュ[1]やファイルシステムキャッシュ[2]のように，メモリを使用することでスループットの向上を図ることが可能となっている．これらの技術を可能にした背景として，メモリの低価格化及びディスクアクセスとメモリアクセスのアクセス性能の差の拡大が挙げられる．メモリの低価格化によってユーザは大容量メモリを容易に使用することができ，ディスクアクセスを抑えることによってスループットの向上を図ることができるようになった．

しかし，メモリ需要の大きい環境下では，メモリが大容量化してもメモリが逼迫し，ページアウトを多発させることがある．その一例として仮想マシン環境がある．仮想マシン環境では，各ゲスト OS の使用するメモリの総容量はホストマシンの物理メモリに対してのオーバーコミットが許可されている．この機能の実現手段として ballooning が提供され，ゲスト OS 間でのメモリの貸し借りが可能となっている[3]．しかし，ballooning が発生すると使用できるメモリが制限されるゲスト OS ではメモリがより逼迫しやすくなる可能性がある．

このような物理メモリ容量が逼迫した状態では，ページングによるメモリの多重化はサービススループットを

劣化させる可能性がある．ページングによるメモリ管理では，ファイルシステムキャッシュのようなディスク上にデータが保証されているメモリ領域ならば解放することができる．しかし，プロセスの獲得したメモリ，特にディスク上にデータが保証されていないメモリはデータが保証されていなければプロセスが異常終了する可能性がある．従ってメモリオーバーコミットを実現するためにはページアウトしなければならない．このため，ページアウトが頻発する状況ではディスクアクセスが多発し，結果としてサービススループットの劣化を引き起こす．

そこで本研究では，データが保証されないメモリ領域を提案する．このメモリ領域では，物理メモリが逼迫するとプロセスが獲得した領域を解放することで，ページングによらないメモリの多重化を実現する．この手法によってデータが保証されないメモリ管理セマンティクスを新たに定義し，ページングを可能な限り回避することでディスクアクセスの発生を抑え，サービススループットの大きな劣化を防ぐことが目的である．

本論文では第2節で現在のメモリ管理セマンティクスとその問題点について議論し，第3節では現在のメモリ管理セマンティクスがはらんでいる問題を解決するために既存研究及びその問題点に議論する．第4節では提案手法について述べ，第5節で提案手法の妥当性を検証する．

## 2 現在のメモリ管理セマンティクス

現在のメモリ管理はOSとプロセスによってそれぞれ行われている．本節ではこれらのメモリ管理セマンティクス及びその問題点について議論する．

### 2.1 プロセスとOSによるメモリ管理

OSの使用するメモリの総容量はホストマシンの物理メモリに対してオーバーコミットが可能になっている．このメモリオーバーコミットはページングによって実現され，プロセスが獲得したメモリ領域は明示的に解放されるまでOSによって保証される．従って，プロセスがメモリを必要とし，ヒープやメモリマップを使用して獲得すると，その領域を明示的に解放，あるいはミドルウェアによるカーベッジコレクションによって不要なメモリ領域と判断され解放されるまでは，確実にアクセスすることができる．

しかし，ファイルシステムキャッシュのようにディスク上に保持されているデータを格納しているメモリ領域はOSによって保証されないことがある．これは，プロセスが必要としたときに解放されていたとしても，再割り当てすることでデータの復元を容易に行い，プロセスが異常終了するようなメモリアクセスを防ぐことができるためである．

これらのことから，現在のメモリ管理セマンティクスにおいて，ディスク上にデータが保持されているか否かが最も重要な要因として挙げられる．つまり，プロセスがメモリ割り当てを行ったデータはディスク上に保持されているとは限らないために，基本的にOSはプロセスが獲得したメモリ領域を解放することはできない．

### 2.2 現在のメモリ管理セマンティクスにおける問題点

前述したようなメモリ管理では，ページアウトが大きなボトルネックとなり，サービススループットの劣化は回避しにくい．これは，解放することのできるメモリ領域はディスク上に保証があるデータのみに限られるからである．

プロセスが獲得したメモリ領域には，ディスク上に保持されているために復元が容易なものとデータが失われると復元が困難なものがある．前者はwebサーバのメモリキャッシュのようにディスクアクセスを減らし，スループットを向上させるためのもので，後者はそれ以外のプロセスが獲得したメモリ領域である．ページングによるメモリ管理では，データの復元が容易に行うことができるメモリ領域であってもページアウトの対象となる．このため，ページアウト・ページインよりも解放・再割り当ての方が高速な場合であってもページアウト・ページインが行われる．

このように，ページングによるメモリ管理ではプロセスにとってページアウトすべきデータとそうでないデータの適当な判別をすることが難しい．従って，ディスク上にデータが保証されていないデータならばページアウトするという判別方法では本質的には不要なディスクアクセスを増やすことになる．この結果としてディスクアクセスを抑制するために獲得したメモリ領域を保証するためにディスクアクセスを行い，サービススループットを損なってしまうという問題点がある．

## 3 関連研究

プロセス内でのメモリ管理によって明示的に不要なメモリ領域を解放し，サービススループットを向上させるという手法は使用できるメモリが制限された場合において非常に有効である．ガーベッジコレクションとメモリの明示的な管理を行った場合の性能差について述べている研究[4]では，メモリが潤沢にあると，ガーベッジコレク

ションの方がスループットをより向上させている．しかし，使用できるメモリが少ない場合においては，明示的なメモリ管理を行った方がページングを抑え，スループットの低下を防いでいる．

従って，メモリが逼迫した状態では明示的にメモリのライフサイクルを管理することによって不要なページアウトを低減させ，性能劣化を防ぐことができるとが言える．しかし，この手法でも使用しているがプロセスの動作に不可欠ではないメモリ領域のライフサイクルの管理を行うことはできない．このため，ページアウト・ページインを行うよりも解放・再割り当てを行う方がスループットの低下を防げる場合の判別ができないという課題は依然として残されている．

## 4 提案手法

これまでに述べてきたように，ページングによるメモリ管理では，スループットを向上させるためのデータをページアウトすることによるディスクアクセスが発生する．この問題を解決するためには，現在のデータが保証されているメモリ管理セマンティクスではなく，プロセスの動作に不可欠でないデータならばディスク上に保証がなくても解放することのできるメモリ管理セマンティクスが必要となる．

そこで本節では，データが保証されないメモリ領域の提案及びその実現手法としてメモリオブジェクト，メモリ管理フレームワークについて述べる．

### 4.1 メモリオブジェクト

データが保証されないメモリ領域を実現するにあたって，最も問題となる点がプロセスが予期しないタイミングで破棄されたメモリ領域の扱いである．プロセスは獲得したメモリ領域は保証されているという前提のもとで動作を行うため，破棄されたはずのメモリ領域を参照する可能性がある．このような場合，通常のメモリ管理ではプロセスが異常終了する．

この問題を解決するために本研究では，プロセスが獲得した破棄される可能性のある領域をオブジェクトとして扱う．以降，この領域のことをメモリオブジェクトとして定義する．メモリオブジェクトは，メモリオブジェクト固有のid，メモリを割り当てるデータへのポインタ，メモリを割り当てたデータの状態，アロケーションサイズ，参照回数，最終参照時刻の属性を持つ．メモリを割り当てたデータの状態には，後述するメモリ管理フレームワークによって解放することができる状態か否か，あるいは既に解放された状態であるかという情報が記載されている．これらの状態遷移は図1に示し，それぞれの状態は以下の通り．

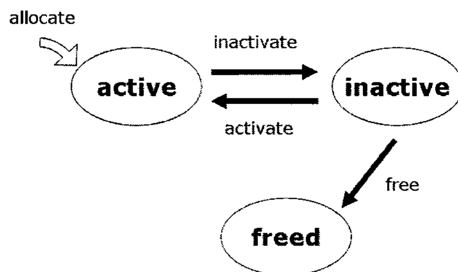


図1 メモリオブジェクトの状態遷移

- active
  メモリを割り当てたデータは使用中であり，メモリ領域は解放されない
- inactive
  メモリを割り当てたデータは使用中でなく，メモリ領域は解放される可能性がある
- freed
  メモリを割り当てたデータは既に破棄された状態である

メモリオブジェクトの生成，解放，状態遷移，演算処理は全てインターフェースとして定義する操作APIを使用して行う．これらの操作APIについては以下に詳細を述べる．

メモリオブジェクトの生成では，割り当てたメモリ領域に対して一意のidを割り振る．idを割り振られると，以降の操作はidをキーにしてのみ行う．割り当て後の状態はinactivateされるまでactiveである．

メモリ割り当てを行ったデータに対しての演算処理は，idを使用して対象となるメモリオブジェクトを特定することで行う．idによって指定されたメモリオブジェクトをactivateし，演算処理中であることを明示し，後述するメモリ管理フレームワークによってメモリオブジェクトが自動解放されないようにする．また，演算処理が終了すると，参照回数，最終参照時刻を更新した後，inactivateし，メモリオブジェクトへの操作が終了したことを明示する．

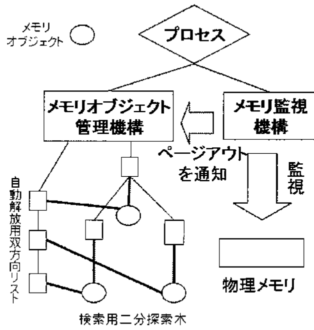


図 2　メモリ管理フレームワークの構造

メモリ割り当てを行ったデータの解放操作中は，メモリオブジェクトを activate し，解放操作が終了するとメモリオブジェクトの状態はは freed へと移行する．

これらの操作によってプロセスの予期しないタイミングでメモリを割り当てたデータが破棄されたとしても，そのメモリ領域への直接の参照を行うことができないためにプロセスが異常終了することを防ぐことができる．

### 4.2　メモリ管理フレームワーク

メモリ管理フレームワークは，メモリオブジェクトの管理を行うフレームワークである．ホストマシンの物理メモリが逼迫し，ページアウトが発生すると管理していたメモリオブジェクトの中で最も優先度の低いものから順次メモリを割り当てていたデータを解放する．この優先度の設定はフレームワークを適用するアプリケーションによって必要なデータが変わるため，LRU, LFU, 割り当てサイズから予め設定する．

この操作を自動で行うことで，ページアウトする必要のないデータのページアウトを防ぎ，ページングによらないメモリの多重化及びデータの保証されないメモリ領域の実現を図る．

### 4.3　実装

メモリ管理フレームワークを構成する要素として，メモリオブジェクト，メモリ使用状況管理機構，メモリオブジェクト管理機構が必要である．これらの構造を図 2 に示す．

メモリ使用状況監視機構は，ホストマシンの物理メモリの使用状況を取得し，ページアウトの発生を検知し，メモリオブジェクト管理機構へと通知する．

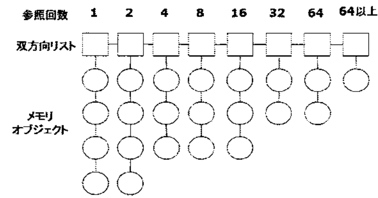


図 3　双方向リスト

メモリオブジェクト管理機構はメモリオブジェクト検索用の id をキーとした二分探索木と優先度順にメモリオブジェクトをソートした双方向リストを持つ．双方向リストはページアウト発生時にアロケートしたデータを順次解放するために使用する．

メモリ使用状況監視機構からページアウト発生通知が来るまで wait する．ページアウト発生通知を受け取ると双方向リストの低優先度メモリオブジェクトから順次割り当てていたデータを解放する．このときの解放する量は予め設定しておく．なお，この双方向リストの先頭が最高優先度，末尾が最低優先度としてリスト操作を行う．

メモリオブジェクトを生成すると，二分探索木及び双方向リストにそれぞれ挿入する．また，プログラム中で明示的にメモリオブジェクトを解放すると，二分探索木，双方向リストからはそれぞれ該当するメモリオブジェクトは削除される．しかし，ページアウト発生時の自動解放が行われる場合では，どのメモリオブジェクトと対応するデータを解放しているのかをメモリオブジェクト監視スレッドは保持しておかなければならないため，二分探索木及び双方向リストからはメモリオブジェクトは削除されない．

双方向リストによって管理されるメモリオブジェクトの優先度は，前述したように LRU, LFU, アロケーションサイズから一手法を選択して決定する．このとき，LRU の場合は操作されたメモリオブジェクトが最も高優先度となるため，双方向リストの先頭へと挿入するのみで操作は終了するため，計算量は O(1) となる．しかし，LFU やメモリの割り当てサイズによって優先度を決定する場合は，ソートが必要となるため，計算量は O(N) となる．この優先度決定手法による計算量の差を抑えるために，LFU と割り当てサイズによる優先度の決定の場合では別のリストをしようする．LFU の場合でのこのリストを図 3 に示す．

この双方向リストは全ノードがそれぞれメモリオブジェクトのリストを持ち，属性としてメモリオブジェクトの参照回数の範囲を持つ．この構造によって参照回数の近いメモリオブジェクトのみをソートすることでリスト操作を終えることができ，計算量を少なくすることができる．優先度選択方法にメモリの割り当てサイズを使用する場合も LFU と同様の処理を行う．

メモリオブジェクトに対する操作 API は libc 標準に相当するものを実装した．メモリオブジェクト生成/解放関数は，malloc()/free(), mmap()/manmap() に相当する関数として定義する．この関数ではメモリを割り当てるデータのサイズがページサイズ未満であればヒープを使用し，ページサイズ以上であればメモリマップを使用する．これはメモリの断片化を防ぐためである．また，その他の演算 API にはメモリ操作関数とファイル I/O 関数を実装した．メモリ操作関数は memcpy() のような mem*関数に相当する関数，ファイル I/O 関数は read(), write() に相当する関数として定義する．

## 5 評価・考察

本節では，提案したメモリ管理機構の妥当性の検証実験を行い，評価，考察する．

### 5.1 評価実験

本実験では，ファイルキャッシュアプリケーションを作成し，提案手法を適用したアプリケーションと適用しないアプリケーション，ファイルシステムキャッシュのサービススループットの差をそれぞれ比較検証する．このファイルキャッシュアプリケーションでは，256KB のファイルを 3072 個 read() し，バッファをキャッシュとして利用する．この処理の後，複数のアクセスパターンでファイルアクセスを行い，/dev/null へと write() する．この処理のスループットを検証し，検証項目は以下のもので行う．

- オーバーヘッドの計測
- 既存手法と提案手法の比較
  - ファイルシステムキャッシュとの比較
  - アプリケーションレベルファイルキャッシュとの比較

#### 5.1.1 オーバーヘッドの計測

オーバーヘッドの計測では，メモリオブジェクトの検索時間，双方向リストのソート時間，これらの操作を除いた操作 API の処理時間を評価項目とする．

メモリオブジェクトの生成関数では二分探索木への挿入操作，演算関数ではメモリオブジェクトの検索操作がそれぞれ必要となる．これらの計算量は共に O(logN) となり，二分探索木への操作がない標準ライブラリの計算量 O(1) と比較してボトルネックとなる．また，メモリオブジェクトの生成関数，演算関数にそれぞれ双方向リストの優先度順のソートが必要となる．このソートはクイックソートで実装されているため，計算量は O(NlogN) となりやはり大きなボトルネックとなる．

メモリオブジェクトの優先度は前述したように LRU, LFU, メモリの割り当てサイズの一手法を選択することで決定する．従って，それぞれの手法で双方向リストへの操作に大きな差がある．LRU では操作されたメモリオブジェクトの優先度はその時点では最も高くなるため，リストの先頭へと挿入するのみでリスト捜査は終了する．しかし，LFU や割り当てサイズではソートしなければならないために LRU に対して処理数が多くなる．このため優先度の決定は，LRU, LFU の二手法でそれぞれ行い，双方向リストのソート時間を計測する．

これらのボトルネックとボトルネックを排除した処理時間を計測することで提案手法のオーバーヘッドを評価する．

#### 5.1.2 既存手法と提案手法の比較

本実験では提案手法を適用するデータをファイルキャッシュとし，ファイルシステムキャッシュ，提案手法を適用していないアプリケーションレベルファイルキャッシュ，提案手法を適用したアプリケーションレベルファイルキャッシュの挙動について検証する．なお，アプリケーションレベルファイルキャッシュではファイルシステムキャッシュの影響は排除する．

検証項目は，意図的にページアウトが発生する状況下でのページアウトの頻度，ファイルキャッシュの参照時間の二項目とする．これらの項目について以下のシナリオでベンチマークを行う．全てのシナリオにおいてアクセスされるファイルの総サイズは 768MB とし，全てのファイルがキャッシュとして取り込まれた状態で使用できる物理メモリを 512MB へと制限する．

**実験 1:** 逐次的なファイルアクセス
使用できる物理メモリを制限した後，全てのファイルに対してシーケンシャルアクセスを試みる

**実験 2:** ランダムなファイルアクセス
使用できる物理メモリを制限した後，どのファイルに

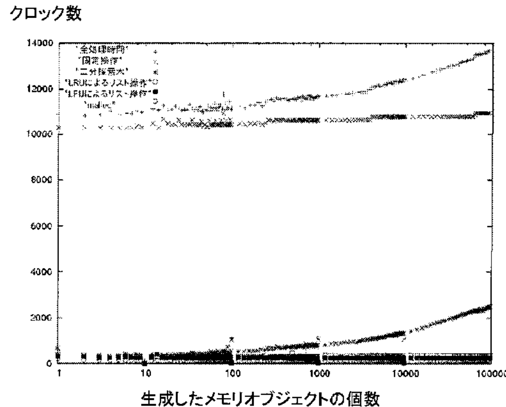


図 4　オーバーヘッド

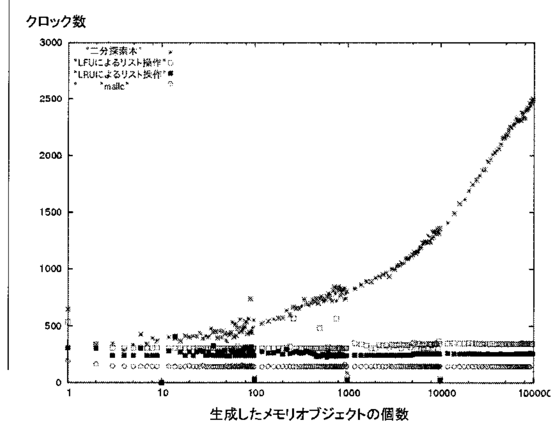


図 5　スケーラビリティのある操作と malloc の比較

対してアクセスが行われるかはランダムに決定する

実験 **3:**偏りのあるファイルアクセス
使用できる物理メモリを制限した後, 特定のファイルに対して多くアクセスがあるようにする. アクセス頻度の偏りは **zipf** の法則に基づく.

これらのシナリオに対してそれぞれのファイルキャッシュ機構の挙動を評価する. 評価に使用したマシンのスペックは, CPU:2.6GHz AMD Opteron252x2, メモリ:1GB DIMMx4 である. また, 評価は使用するメモリを 1GB に制限し, 実験を行った.

## 5.2　結果・考察

### 5.2.1　オーバーヘッドの計測結果

オーバーヘッドの計測結果を図 4 に示し, スケーラビリティのある操作と malloc() を比較した結果を図 5 に示す.

グラフからわかるように, オーバーヘッドの大部分はスケーラビリティのない操作によって占められている. また, スケーラビリティのある操作の中で最も処理時間の遅いものは二分探索木への操作である.

スケーラビリティのない操作内では, 主にメモリオブジェクト id の生成を行っている. 現在の実装では生成したメモリオブジェクトの総数をキートとした hash 値によって id は生成される. この実装では非常にオーバーヘッドが大きくなるために別の手法によって id を生成する必要があると言える.

二分探索木への操作は, 生成したメモリオブジェクトの総数が多くなるほどオーバーヘッドが大きくなっている. これは, 二分探索木が全てのメモリオブジェクトを一括して管理していることに起因する. このため, このオーバーヘッドを小さくする, あるいは一定にするためには, メモリオブジェクトの総数が一定値を超えた場合には新たな管理用のデータ構造を用いるといった複数の管理手法が必要となる.

LRU によるリスト操作は, スケーラビリティはあるが, メモリオブジェクトの個数に関わらず常に処理数は一定となる. また, その処理はリストの末尾に追加する操作のみのため, オーバーヘッドは小さい. LFU によるリスト操作は LRU と比較して若干オーバーヘッドが大くなっているが, 大きな差異はない. これは, 一般に参照回数の多いほど該当するファイルが少なくなることに加え, 参照回数の少ないものほどソートする処理が少ない実装となっていることに起因する.

これらのことから, ページアウトが発生していない状況では提案手法は大きなオーバーヘッドを生じさせる結果となった. 次節ではページングが発生し, 通常のメモリ管理手法ではサービススループットが劣化する場合において, 提案手法の評価を行っている.

### 5.2.2　既存手法と提案手法の比較結果

前節で述べた実験 1, 2, 3 において, ファイルシステムキャッシュ及びアプリケーションレベルファイルキャッシュ

の平均アクセス時間及びページアウトが発生した場合の総ページアウトサイズを計測し，その実験結果を表 1, 2, 3, 4 に示す．なお，表内の FS, App, FW はそれぞれファイルシステムキャッシュ，アプリケーションレベルファイルキャッシュ，提案手法を適用したアプリケーションレベルファイルキャッシュとする．

表 1　物理メモリを制限しない場合

| 実験方法 | 平均処理時間 [usec] | ページアウトサイズ [MB] |
|---|---|---|
| FS | 637 | 0 |
| App | 381 | 0 |
| FW | 884 | 0 |

表 2　実験 1 結果

| 実験方法 | 平均処理時間 [usec] | ページアウトサイズ [MB] |
|---|---|---|
| FS | 982 | 0 |
| App | 1809 | 239.128 |
| FW | 1277 | 15.916 |

表 3　実験 2 結果

| 実験方法 | 平均処理時間 [usec] | ページアウトサイズ [MB] |
|---|---|---|
| FS | 891 | 0 |
| App | 1692 | 228.991 |
| FW | 1039 | 15.623 |

物理メモリを制限しない環境下では，App の処理速度が最も速く，FW の処理速度が最も遅くなった．これは，メモリ管理フレームワークのオーバーヘッドに起因する．また，App と FS を比較する．FS では，アクセスのあったファイルと対応するファイルキャッシュを read() しなければならない．しかし，アプリケーションレベルでフィルキャッシュを保持していると，アクセスのあったファイルを再び read() する必要がなく，処理が少なくなる．このため，App が最も処理速度が速い結果となった．

表 4　実験 3 結果

| 実験方法 | 平均処理時間 [usec] | ページアウトサイズ [MB] |
|---|---|---|
| FS | 823 | 0 |
| App | 1630 | 201.276 |
| FW | 1019 | 13.371 |

実験 1 では，FS の処理速度が最も速く，App が最も遅くなった．FS では全てのファイルキャッシュをファイルシステムが管理しているため，ディスク上に保証され，解放可能なデータによって物理メモリが使用されているために，物理メモリが逼迫するとファイルキャッシュを解放することで空きメモリ容量を増やすことができる．しかし，物理メモリを制限していない場合と比較すると，ファイルキャッシュが解放されたことにより，キャッシュのヒット率が下がり，アクセスのあったファイルを再び read() しなければならず，ディスクアクセスが増加し，処理時間は低下している．

App では，アプリケーションレベルファイルキャッシュのため，OS からはプロセスが割り当てたメモリはディスク上に保証されているかは判別できない．このため，プロセス内で確保したファイルキャッシュをページアウトしなければならず，ディスクアクセスが多発し，処理速度の低下を招く．

FW では，ページアウトが発生するとファイルキャッシュが解放され，空きメモリ容量を増やす．この操作により，後続のページアウトを回避することができるが，FS と同様にアクセスのあったファイルを再び読み込まなければならない．ページアウトを完全に回避することができないことに加え，ファイルを再読み込みしなければならないため，FS と比較すると処理速度は遅くなり，後続のページアウトを防ぐことができ，結果としてディスクアクセスを低減できているために App よりも処理速度は速い．

実験 2 では，実験 1 と比較すると全ての手法において処理速度が向上している．シーケンシャルアクセスの場合では，ファイルキャッシュを確保した順番とファイルアクセスの順番が同様になるため，FS ではキャッシュのヒット率が極めて低くなる．しかし，ランダムアクセスの場合は，ファイルキャッシュを確保した順番とファイルアクセスの順番は全く違うものとなるため，解放されていな

いファイルキャッシュへとアクセスされる確率が上がり，キャッシュヒット率が向上する．App の場合は，ページアウトされているファイルキャッシュへとアクセスする確率が低くなり，実験 1 と比較するとディスクアクセスを低減することができる．また，FW では，実験 1 よりも解放されたファイルキャッシュへとアクセスされる確率が低くなるためにファイルの再読み込みが少なくなり，ディスクアクセスを抑えている．

実験 3 では，アクセスのあるファイルが偏っているために，実験 2 よりも更にキャッシュヒット率が向上する．このため，App, FW ではページイン・ページアウトの回数が減少し，処理速度の向上が見られる．

これらのことから，いくつかのアクセスパターンにおいて，物理メモリが逼迫した環境下ではサービススループットを向上させるためのメモリ領域を解放することによってサービススループットの劣化を防ぐことができると言える．

## 6 まとめと今後の課題

本論文では，ページイン・ページアウトによらないメモリ管理機構の提案及び評価を行った．その結果として，サービススループットの向上を図るためのデータを解放することでページアウトが多発する環境下でのサービススループットの劣化を抑えることができた．これにより，ページアウトが発生すると解放されるメモリ領域の有効性を確かめることができた．

本論文で提案したページングによって保証されないメモリ領域は，その性質上復元が容易に行うことのできないデータに対しては適用することができない．従って，サービススループットの向上に使用するデータのみに有効である．しかし，空きメモリ容量が多く，ページアウトが発生しない環境では，オーバーヘッドによってサービススループットを損なってしまうことがある．このオーバーヘッドを小さくすることが必要である．

また，第 1 節で述べたように，本研究では仮想マシン環境のような OS が使用できるメモリ容量が変動する環境への適用が目標である．本論文では擬似的にこのメモリ需要が変動する環境を作ったが，実環境では未評価である．従って，実環境への適用を行い，挙動を評価することが今後の課題と言える．

## 参考文献


1) A. Chankhunthod, P. Danzig, and C. Neerdaels, A hierarchical Interact object cache. In Proce edings of J996 USENiX Teclmical Conference, Jan 1996.

2) Silvano Maffeis, Cache management algorithms for flexible fil esystems, ACM SIGMETRICS Performance Evaluation Review, Dec 1993

3) Carl A. Waldspurger. Memory Resource Management int VMWare ESX Server, Palo Alto, Dec 2002

4) Matthew Hertz, Emery D. Berger, Quantifying ther Performance of Garbage Collection vs. Explicit Memory Management, OOPSLA'05, Oct 2005

5) Zhifeng Chen, Yhang Zhang, Yuanyuan Zhou, Heidi Scott, and Berni Schiefer, Empirical Evaluation of Multi-level Buffer Cache Collaboration for Storage System, SIGMETRICS'05, June 2005