データベースシステム 93－2
システムソフトウェアと
オペレーティング・システム 59－2
（1993．5．27）

# 分散型ストレージサーバ WAKASHI の実装および評価

小柳雅彦　白光一　天野浩文　牧之内顕文

九州大学工学部情報工学科
〒812 福岡市東区箱崎 6-10-1

WAKASHI は分散共有永続ヒープを提供するストレージシステムである．一つのサーバが集中的に処理を行なう従来の実現方式には，サーバ自身がボトルネックになる等の問題があった．本稿ではこれらを解決するために，各サイトにサーバを配置する分散型サーバを提案し，その実現方法を述べる．また，従来の集中型サーバと本研究で実装した分散型サーバを比較する形で，理論式を用いた性能評価を行なう．

# Implementation and Evaluation of a Distributed Storage Server WAKASHI

Masahiko Koyanagi　Guangyi Bai　Hirofumi Amano　Akifumi Makinouchi

Department of Computer Science and Communication Engineering
Kyushu University
6-10-1 Hakozaki Higashi-Ku Fukuoka, 812 Japan

WAKASHI is a storage system which provides distributed shared persistent heaps. WAKASHI has been implemented as a centralized server which manages DSPHs intensively. But there are several problems about it such as the server itself becomes the bottleneck. This paper describes the design and implementation of new distributed solution that can allocate any number of servers on any number of machines. The performance evaluation of the distributed server by contrast with the centralized one is also described in this paper.

## 1 まえがき

WAKASHI は分散共有永続ヒープ (distributed shared persistent heap, DSPH) を提供するストレージシステムである [BM92, BT92]. 図 1のように, 複数のワークステーションをネットワークで結合した分散環境において, タスクの仮想メモリ空間に二次記憶上のファイルと 1 対 1 に対応した領域を設け, これを DSPH と呼ぶ. DSPH 領域では, 応用プログラムでファイル入出力を行なわなくてもデータの保存・再利用ができるだけでなく, 通常のヒープ領域において揮発データを扱う場合と同様にファイル上の永続データを扱うことができる. また, ローカルファイルだけでなくリモートファイルのマッピングも可能である. 主記憶・二次記憶間のデータの一貫性, および, 異サイト間のデータの一貫性は WAKASHI サーバが統一的に管理する.

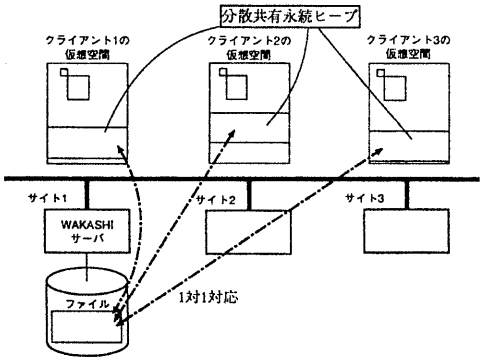


図 1: 分散共有永続ヒープ

以前の実現法 [BT92] では, 1 つの WAKASHI サーバが全サイトのクライアントに対して集中的にサービスを提供する形をとっていた. しかしこのような集中型サーバには, サイト間の不平等性やサーバ自身がボトルネックになる等の問題がある. また, サイト間のページの移動の際, 必ずサーバのあるサイトを経由しなければならないので通信量が増加する. これらの問題を解決するため, 本論文では新しく分散型サーバを実装した. サーバの分散化は分散共有メモリの実現法の一つで [FB88, LH89], すべてのサイトに各々サーバを割り当て, サーバ間の通信によってメモリコヒーレンスを保持するものである. 我々はこれを WAKASHI サーバに適用して, 集中型サーバにあった問題点の解決を図った.

本稿では, WAKASHI サーバの実現法, 集中型サーバのアルゴリズム, および新しく実装した分散型サーバの詳細を述べる. また, 本研究で実装した分散型サーバと従来の集中型サーバとを比較する形で, 性能評価を行ない, 結果を考察する.

## 2 WAKASHI サーバの実現法

WAKASHI サーバは Mach の外部ページャとして実現されている [BM92]. Mach の仮想メモリシステムでは, ユーザ定義の仮想メモリ管理タスクによって主記憶と二次記憶とのページングを管理することができ, このタスクを外部ページャと呼んでいる [IS92]. 主記憶上にないデータへのアクセスは, Mach カーネルによって自動的に外部ページャに対するページング要求に変換される. この要求を受けて外部ページャは実際のページイン, ページアウトを実行する. このような外部ページャは Mach のメモリオブジェクト毎に定義できるので, オブジェクト毎に異なったページング方式の実装や, ネットワークを介してページングを行なうシステムの構築などをユーザレベルで行なうことが可能となっている.

WAKASHI の特徴は, メモリマップドファイルと分散共有メモリを結合しているところである. メモリマップドファイルとは, 二次記憶上のファイルをタスクの仮想空間に直接マップすることで, ファイルを仮想空間上の連続領域のバイト列としてアクセスすることを可能にする方式である. その特徴は

- ファイル上の永続データとプログラムが同一の仮想空間に存在するため, 通常の揮発データと同様に永続データを操作できる.
- ファイル上のデータ形式と主記憶上のそれとが同一であるため, データ形式変換のオーバヘッドがない.

などである.

分散共有メモリは, ローカルメモリを持つプロセッサ群をネットワークで相互接続したシステム上で, ソフトウェアによって各プロセッサに仮想的な共有メモリを持たせる方式である. 基本的な実現方法は, 各プロセッサのローカルメモリに共有する仮想メモリをマップするもので, プロセッサ間のページの移動とコピーによって, メモリコヒーレンスを保持する.

WAKASHI では以上の両方式を結合して, 二次記憶上のファイルを分散共有メモリのページング領域として使用することにより, DSPH を実現している. 分散共有メモリのためのサイト間の一貫性制御と, メモリマップドファイルのための主記憶・二次記憶間の一貫性制御を統一的に扱うため, WAKASHI サーバは DSPH を Mach のメモリオブジェクトとしてとらえ, 異サイト間および主記憶・二次記憶間のページングを行なう.

サーバの機能は次のように大きく 3 つに分けられる.

(1) インタフェース

WAKASHI のインタフェースは Mach の外部ページャインタフェースとサーバインタフェースからなる. サーバインタフェースは主にメモリオブジェクトの生成・破壊に関する新しく定義したインタフェース群である. 表 1にこれらのインタフェースをまとめている.

表 1: WAKASHI のインタフェース

ユーザがサーバを呼び出すインタフェース

- create_memory_object()
  新しくメモリオブジェクトを生成し，それを表すポートを返す．
- destroy_memory_object()
  使用済みのメモリオブジェクトを破壊する．

ユーザがカーネルを呼び出すインタフェース

- vm_map()
  タスクの仮想空間にメモリオブジェクトをマップする．
- vm_deallocate()
  与えられたアドレス範囲のマッピングを解放する．

カーネルがサーバを呼び出すインタフェース

- memory_object_init()
  メモリオブジェクトの初期化をサーバに要求する．
- memory_object_data_request()
  物理的なキャッシュにないデータをサーバに要求する．
- memory_object_data_unlock()
  メモリオブジェクトの特定のエリアに対して，指定したアクセス方法を許すことをサーバに要求する．
- memory_object_data_write()
  変更したページを主記憶からページアウトする．
- memory_object_lock_completed()
  メモリオブジェクトのアクセス方法を制限する処理が終了したことをサーバに通知する．

サーバがカーネルを呼び出すインタフェース

- memory_object_set_attributes()
  メモリオブジェクトに属性を設定する．
- memory_object_data_provided()
  カーネルにデータを供給する．
- memory_object_lock_request()
  カーネルに対してキャッシュの管理を要求する．

**(2) データの永続性管理**

WAKASHI では，主記憶と二次記憶とのページングによりデータの永続性を保持する．データは，それを使用するタスクの存在期間とは無関係にファイル中に保存され，他のタスクによって再使用が可能である．このような性質を持ったデータを，永続データと呼ぶ．主記憶上にないデータのファイルからの読み出し (ページイン)，使用済みデータのファイルへの書き込み (ページアウト) は WAKASHI サーバが自動的に行なうため，クライアントは明示的に要求を出す必要はない．

**(3) クライアント間のデータの一貫性管理**

WAKASHI はサイト間のページングにより，シングルライタ／マルチリーダ方式でデータの一貫性を管理する．データはクライアントのローカルメモリにコピーされ，使用される．DSPH に対する書き込みが発生すると，WAKASHI サーバは書き込みを要求したクライアント (ライタ) 以外のクライアントが持つ全てのページを無効にし，ライタのアクセスが終了するまで他のクライアントのアクセス要求は待たされる．ライタがデータを書き戻すとそれを待っていた次のクライアントにデータがコピーされ，DSPH へのアクセスが続行される．これらのクライアント間のページングに対する要求は，Mach カーネルによってサーバへ送られ，自動的に処理されるため，ユーザは他のクライアントのアクセスを意識する必要がない．

## 3 集中型サーバのアルゴリズム

集中型サーバでは *owner*, *state*, *readers*, *writers* の 4 つのフィールドを持つ構造体により DSPH の各ページを管理している．それぞれのフィールドの持つ意味は次の通りである．

- *owner*では最新のページを持つクライアントを登録している．すなわち，現在ページにライトアクセスを行なっているクライアントである．どのクライアントもライトアクセスしていない場合は，WAKASHI サーバがそのページの *owner*になり，この場合に限り複数のクライアントがリードのみ可能のコピーを持つことができる．
- *state* ではページ状態を保持する．WAKASHI ではページ状態を

  **Read 状態：** どのクライアントもライトアクセスしていない状態．複数のクライアントがリードアクセスしている可能性がある．サーバは最新のコピーを持っている．初期状態．

  **Write 状態：** 1 つのクライアントがライトアクセスしていて，かつ他のどのクライアントもそのページがページアウトされるのを待っていない状態．サーバはページの有効なコピーを持たない．

  **ReadWait 状態：** 1 つのクライアントがライトアクセスしていて，かつ 1 つ以上のリーダがそのページを待っている状態．サーバは有効なコピーを持たず，*owner*に対してページを戻すよう要求している．

**WriteWait 状態：** 1つのクライアントがライトアクセスしていて，かつ1つ以上のライタがそのページを待っている状態．さらに複数のリーダが待っている可能性がある．サーバは有効なコピーを持たず，*owner*に対してページを戻すよう要求している．

の4状態にわけ，クライアントからの要求によって状態を変化させる．

- *readers*では，ページに現在リードアクセスしているクライアント，およびリードアクセスするためにページが供給されるのを待っているクライアントを全て登録している．
- *writers*では，ライトアクセスするためにページが供給されるのを待っているクライアントを全て登録している．

カーネルからのページング要求は *read_fault()*,*write_fault()* , *pageout()* の3つの関数によって処理される．図2，図3および図4にこれらの関数の疑似コードを示す．サーバ内のその他の関数としてはメモリオブジェクトの生成，破壊，および初期化の各関数がある．

```
read_fault(page, client)
  switch ( page->state ) {
  case Read:
    memory_object_data_provided(client)
    break
  case Write:
    page->state = ReadWait
    memory_object_lock_request(page->owner,
      CLEAN(page), server_self)
    break
  default: /* just enqueue */
  }
  set_add(page->readers, client)
```

図 2: *read_fault()*の処理

```
write_fault(page, client)
  switch ( page->state ) {
  case Read:
    set_remove(page->readers, client)
    forall(readers)
(1)   memory_object_lock_request(reader,
        FLUSH(page), NONE)
    page->readers = empty_set
(2)
    page->state = Write
    page->owner = client
    if (needs_data)
      memory_object_data_provided(
        page->owner)
    else
      memory_object_lock_request(
        page->owner, UNLOCK(page), NONE)
    break
  case Write:
    memory_object_lock_request(page->owner,
      CLEAN(page), server_self)
  case ReadWait:
    page->state = WriteWait
  case WriteWait:
    enqueue(client, page->writers)
  }
```

図 3: *write_fault()*の処理

```
pageout(page, client, data)
(3)
  switch ( page->state ) {
  case Read:
    return   /* never happens */
  case Write:
    save(data)   /* true pageout */
    page->state = Read
    page->owner = server_self
    break
  case ReadWait:
    save(data)
    forall(readers)
      memory_object_data_provided(reader)
    page->state = Read
(4) page->owner = server_self
    break
  case WriteWait:
    save(data)
    page->owner = dequeue(page->writers)
    memory_object_data_provided(page->owner)
    if (!page->writers)
       if (page->readers)
         page->state = ReadWait
       else
         page->state = Write
    if (page->readers || page->writers) {
      memory_object_lock_request(page->owner,
        CLEAN(page), server_self)
    }
  }
```

図 4: *pageout()*の処理

### 3.1 集中型サーバの問題点

集中型サーバの改良の動機となった問題点を次に示す.

- ネットワークが拡大しクライアント数が増加すると，多くの要求が一つのサーバに集中して，サーバがボトルネックになる.
- ローカルメッセージとリモートメッセージのコストが大きく異なるので，サーバの位置するサイトと他のサイトが対等でない．プログラマはこの問題を考慮してプログラミングする必要がある.
- サイトから他のサイトへページを移動するには，集中型サーバでは4つのメッセージ(クライアントAからサーバへページ要求，サーバから *owner* へページアウト要求，サーバへページアウト，サーバからクライアントAへページ供給)が必要である．もし *owner* からクライアントAへ直接ページを送れば，通信量を減らすことができる.

## 4 分散型サーバ

集中型サーバにはいくつかの問題があった．これらの問題を解決，あるいは軽減するため，図5のようにWAKASHIサーバを各サイトに分散することが考えられる．このように，全てのサイトにサーバを置くことで，クライアントは全てローカルにサーバを持つこととなり，サイト間の不平等は解消される．また，各サーバはローカルサイトのメモリオブジェクトのみを管理し，リモートサイトのメモリオブジェクトはそのサイトのサーバが要求を受けて管理するので，サーバのボトルネックの軽減が期待できる．さらに，サーバを分散化することによって，処理効率向上のためのさまざまな手法を実装できるようになる.

### 4.1 分散化の方法

サーバを分散化する方法は非常に単純である．3節で説明した集中型サーバのアルゴリズムを各サイトにコピーして使用し，その際，リモートサイトのWAKASHIサーバをリモートクライアントとして扱う．つまり，図2〜図4の疑似コードで *client* の意味するところは，ローカルクライアントか，もしくは他のサーバとなる．分散型サーバでの主な変更点を次に示す.

(1) フォワーディングのサポート

分散型サーバと集中型サーバとの最も大きな違いは，*owner* の扱いである．分散型サーバでは *owner* は常に正しい値を保持している必要はなく，各サーバの *owner* をたどって最終的に正しい *owner* にたどり着ければ十分である．さもないと，*owner* が変わるたびに，メモリオブジェクトをコピーしている全てのサーバの *owner* を更新する必要が生じ，サーバ間の通信量が増大してしまう．*owner* をたどることで，あるサーバのページング要求が，別のサーバによって第3のサーバへと送られることをフォワーディングと呼ぶ．フォワードされた要求は，正しい *owner* によって処理された後，要求を出したサーバに直接返答される．これによってページをサイトから他のサイトへ移す時に生じる通信量を減らすことができる.

(2) サーバ間インタフェースの作成

分散型サーバは他のサーバからの *memory_object_lock_request()* 及び *memory_object_data_provided()* を受け取ることができる．*memory_object_lock_request()* は他のサイトで生じたページフォールトの通知として扱われる．メッセージを受け取ったサーバがデータをすぐに準備できれば，*memory_object_data_provided()* によってデータを供給するが，そうでなければ，*memory_object_lock_request()* は他のサーバにフォワードされる．*memory_object_data_provided()* は他のサーバからのページアウトとして *pageout()* で処理される．待たされていたクライアントは *pageout()* によってデータを供給され，中断していた処理を再開する.

(3) *replicate_memory_object()* の作成

分散型サーバではメモリオブジェクトの生成を1つのサーバで行ない，他のサイトのサーバはそのメモリオブジェクトをコピーして使用する．コピーを行なうためのインタフェースが *replicate_memory_object()* である．*replicate_memory_object()* によってメモリオブジェクトをコピーしたサーバでは，ページの初期状態は(Readではなく) Write状態になる．*owner* にはメモリオブジェクトをコピーした元のサーバがセットされる．これによって，クライアントが最初にDSPHのページにアクセスし，ページフォールトが発生した時，最新のデータを持つサーバにデータ要求を出すことができる(そのサーバがすでにデータを持っていない場合は，要求はフォワードされる).

図2〜図4の3つの関数のうち，サーバの分散化に際して変更した部分を図6に示す.

```
(1) memory_object_lock_request(reader,
      FLUSH(page),
      is_server(reader) ? client : NONE)

(2) if (page->owner != server_self) {
      memory_object_lock_request(
        page->owner, CLEAN(page),
        server_self)
      enqueue(page->writers, client)
      page->state = WriteWait
      return
    }
```

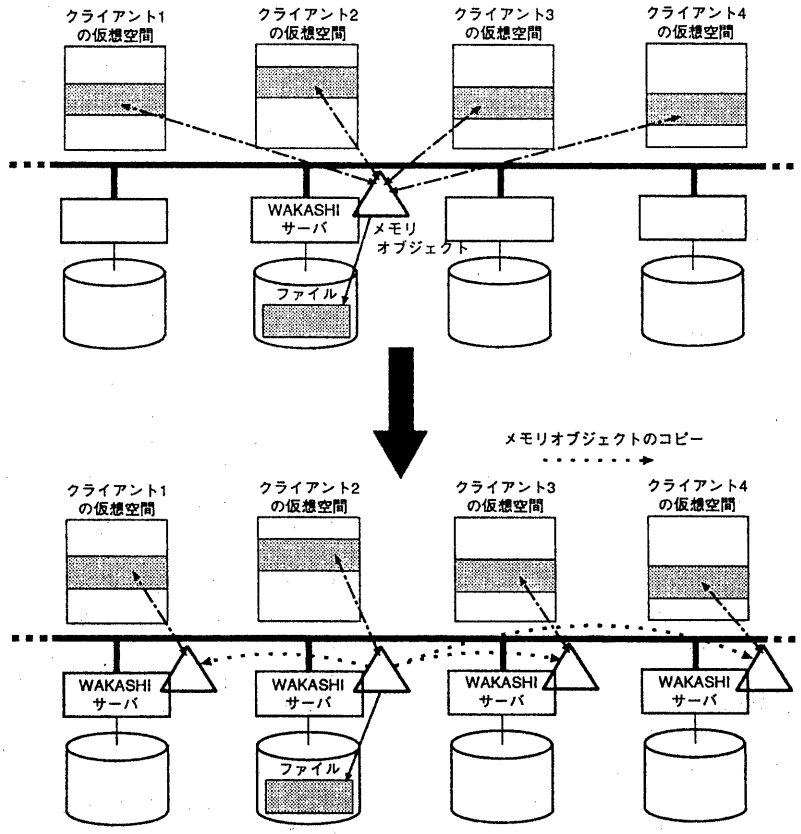


図 5: サーバの分散化

```
(3) if (client != page->owner
       && !hinted(page))
     page->owner = client
   hinted(page) = FALSE

(4) if (!is_server(client))
       page->owner = server_self
```

図 6: フォワーディングのための変更点

分散型サーバにおいても *read_fault()* は集中型サーバと同じアルゴリズムを使用する．しかし，*write_fault()* 及び *pageout()* には，主に *owner* の値が常に正しくないことに起因する若干の変更部分がある．図 6 (1) では，ロックされるリーダが他のサーバであった場合，新しい *owner* の情報が与えられる．(2) では，他のサーバが *owner* かどうかのチェックが追加される．他のサーバが *owner* であった場合には，クライアントのライトフォールト要求は待ち行列につながれ，*owner* からのページアウトを待つことになる．*pageout()* では，(3) で *owner* の値が正しくない場合の処理が追加される．サーバはライトフォールトをフォワードした時，最終的な *owner* に対する *hint* を受け取るので，ここではそれを利用している．(4) では，サーバがリードアクセスの許可だけを受け取った場合 *owner* にはならないので，そのチェックが追加されている．

分散型サーバでは，図 7に示す 2 つの関数が追加される．*page_fault()* はフォールト要求をフォワードするかどうか判断する．他のサーバから受け取った *memory_object_lock_request()* は初め *page_fault()* で処理され，必要ならばフォワードされる．フォワードされない場合は，*fault_type* によって *read_fault()* もしくは *write_fault()* が実行される．ページング要求がフォワードされないのは

- サーバもしくはサーバのローカルなクライアントがページの *owner* である場合
- サーバが現在の *owner* に対してライトアクセスのためにデータを要求している場合 (WriteWait 状態)
- 要求がリードフォールトで，かつサーバがリードのみ可能のコピーを持っている (Read 状態) かこれから持とうとしている (ReadWait 状態) 場合

のいずれかである．言い替えれば，要求はサーバがそのページに全く関係ない場合に限り，フォワードされる．フォワードされた場合，要求がライトフォールトの場合に限り，*owner*を要求を出したサーバ(すなわち，最終的にデータを受け取るサーバ)に書き換え，ライトフォールトがフォワードされたことを*hint*に保持する．

*invalidate_page()*はリードオンリーのページを無効にする．あるサイトでライトフォールトが起こった場合，そのページ範囲のデータを持つ全てのクライアント(リーダ)に対して，*memory_object_lock_request()*を発行して，ページをロックする必要があるが，他のサーバがリーダになっていた場合には，*memory_object_lock_request()*を受けて，*invalidate_page()*が実行される．具体的には，(1)および(5)で*reader*が他のサーバだった場合に，そのサーバで実行される．*invalidate_page()*では，クライアントのページをロックした後，*state*をWrite状態にし，*owner*をライトフォールトを起こしたサイトのWAKASHIサーバにセットする．

```
page_fault(page, who, fault_type)
  if ((page->owner == server_self) ||
      !is_server(page->owner) ||
      (page->state == WriteWait) ||
      ((fault_type == READ) &&
       (page->state != Write))) {
    if (fault_type == READ)
      read_fault(page, who)
    else
      write_fault(page, who)
    return
  }
  /* Forward */
  send_page_fault(owner, who, page)
  if (fault_type == WRITE) {
    page->owner = who
    hinted(page) = TRUE
  }

invalidate_page(page, owner)
  if (page->state != Read)
    return
  forall(readers)
(5)  memory_object_lock_request(reader,
       FLUSH(page), NONE)
  page->state = Write
  page->owner = owner
```

図 7: 追加される関数

## 5 性能評価

本節では，従来の集中型サーバと本研究で実装した分散型サーバの性能について，数式を用いた比較評価を試みる．

### 5.1 評価方法およびパラメータ

$s$のサイトから構成されるWAKASHIシステムにおいて，クライアントがDSPH領域にアクセスしてページフォールトが起こった場合のページング処理に要する平均通信量を$s$の関数として導出し，集中型サーバと分散型サーバの比較を行なう．サーバの性能はクライアントプログラムのアクセスパターンによって大きく影響されるが，ここでは以下のような条件を仮定する．

- 着目しているページをマップしているサイト数は$s$である
- クライアントのリード／ライト比は一定である
- ローカルメッセージのコストはリモートメッセージのそれと比較して非常に小さいので無視し，リモートメッセージのコストのみを計算に入れる
- メッセージの衝突による通信の待ち時間，およびサーバでのスケジューリングによる待ち時間の計算はここでは省略する

これらの条件の下で，理論式の導出に用いたパラメータを表2に示す．表2のパラメータを用いて，クライアントの1回のページフォールトに要する通信量$C_c$，$C_d$を算出する．添字の$c$および$d$は，それぞれ集中型サーバおよび分散型サーバを表す．

表 2: 性能評価に用いたパラメータ

| |
|---|
| $C$：DSPHに対するアクセスによって発生するページフォールトに要する平均通信量．$s$の関数として導出する． |
| $d$：データを含まないリモートメッセージの送受信にかかる時間． |
| $D$：データブロックを含むリモートメッセージの送受信にかかる時間．ページサイズを8kバイトとすると，$D/d \fallingdotseq 20$程度である． |
| $s$：サイト数 |
| $r$：クライアントのリード／ライト比．r回のリードアクセスに対し，1回のライトアクセスが起きると仮定する |

### 5.2 式の導出

次に1回のページフォールトに要する平均コスト$C_c$，$C_d$の算出を行なう．

集中型サーバのコスト $C_c$ の算出

1) ローカルクライアントがページフォールトを起こした場合のコスト $x$

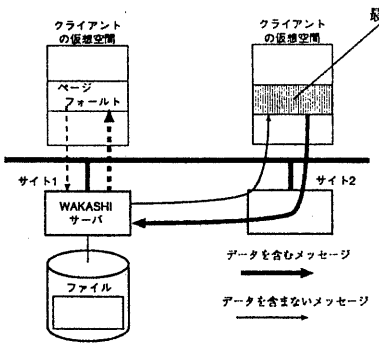


図 8: 集中型サーバのローカルクライアントでページフォールトが発生した場合

サーバがページを持っている確率を $h$ とすると，$h$ はそのページが Read 状態である確率に等しい．また，その時のリーダ数の平均を $R$ とすると

$$x = h \cdot \frac{1}{r+1} \cdot (R \cdot d) + (1-h) \cdot (D+d) \quad (1)$$

$1/(r+1)$ はフォールトがライトフォールトである確率を表す．式 (1) の第 1 項は Read 状態においてリーダにページを無効化する要求を出すコストである．第 2 項は Read 以外の状態で owner にページアウト要求を出すコスト，および実際のページアウトに要するコストである．

2) リモートクライアントがページフォールトを起こした場合のコスト $y$

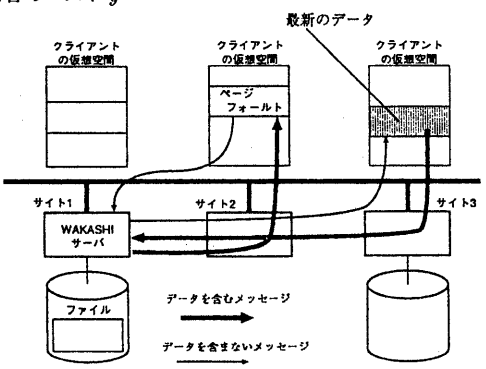


図 9: 集中型サーバのリモートクライアントでページフォールトが発生した場合

$$\begin{aligned} y &= h \cdot \left( \frac{r \cdot (D+d)}{r+1} + \frac{D+d+R \cdot d}{r+1} \right) \\ &\quad + (1-h) \cdot (2D+2d) \\ &= h \left( D+d+\frac{R}{r+1} d \right) \\ &\quad + (1-h)(2D+2d) \end{aligned} \quad (2)$$

式 (2) 第 1 項は Read 状態においてデータの要求，供給にかかるコスト，およびライトフォールト時リーダの持つページを無効化するコストを表す．第 2 項は Read 以外の状態でのクライアントのデータ要求，owner に対するページアウト要求，実際のページアウト，およびクライアントへのデータ供給のそれぞれにかかるコストを表す．

1), 2) より集中型サーバにおいて，1 回のページフォールトに要する平均コスト $C_c$ は

$$C_c = \frac{1}{s} \cdot x + \left(1 - \frac{1}{s}\right) \cdot y \quad (3)$$

となる．式 (3) に (1)，(2) を代入して

$$C_c = 2(D+d) + \frac{hR}{r+1} d - h(D+d) - \frac{1}{s}(D+d) \quad (4)$$

となる．

分散型サーバのコスト $C_d$ の算出

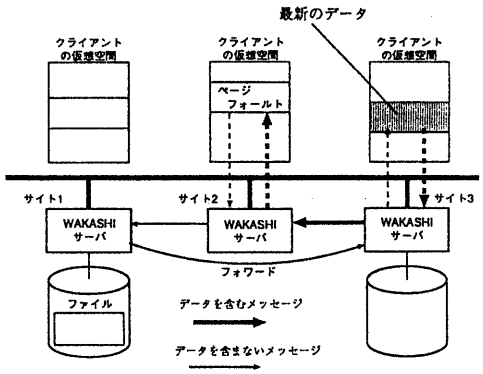


図 10: 分散型サーバでページフォールトが発生した場合

ページフォールト要求がフォワードされる回数の平均を $f$ とする．

$$\begin{aligned} C_d &= d + f \cdot d + D + \frac{R'}{r+1} \cdot d \\ &= D + d + fd + \frac{R'}{r+1} d \end{aligned} \quad (5)$$

$R'$ は，ライトフォールト時リーダのいるサイト数の平均を表す．式 (5) 第 2 項はページフォールトが起きたサイトのサーバから owner へのデータ要求にかかるコストを表し，第 3 項は (もし必要なら) それをフォワードするコストを表す．第 1 項はこれらの要求に対してデータを供給する時にかかるコストである．最後の第 4 項はライトフォールト時，リーダに対してページを無効にする要求を出すコストを表す．

## 5.3 結果

式を簡潔にするため，次のようなパラメータの置換えを行なう．

- $D/d = 20$
- $h = \left(\frac{r}{r+1}\right)^s$ :
  集中型サーバでRead状態である確率$h$は，全サイトでライトアクセスが起こらない確率に等しいとして，近似した．
- $R = \frac{s-2}{2}$ :
  集中型サーバでRead状態である時のリーダ数の平均$R$は，最大値$(s-2)$と最小値0の平均で近似した．
- $fd = \frac{C_d}{10}$ :
  $r = 0$の場合(同じページに全てのクライアントがライトアクセスする場合)のフォワードされるページフォールトは，全体の10%であるため[FB88]，ここではフォワーディングにかかるコストを全体の1/10としている．
- $R' = \frac{(s-f-2)r}{r+1}$ :
  分散型サーバで，ページフォールトが起きた時のリーダのいるサイト数の平均$R'$は，クライアントがリードアクセスする確率
  $r/(r+1)$を最大値$(s-f-2)$だけ足した和で近似した．

これらを式(4)，(5)へ代入して整理すると，

$$\frac{C_c}{d} = 42 - \frac{21}{s} + \left(\frac{s-2}{2(r+1)} - 21\right)\left(\frac{r}{r+1}\right)^s \quad (6)$$

$$\frac{C_d}{d} = \frac{10(r+1)^2}{9(r+1)^2 + r}\left(21 + \frac{r(s-2)}{(r+1)^2}\right) \quad (7)$$

となる．式(6)および(7)を$r=1$，$r=3$，$r=5$，$r=10$の4通りで次のようにグラフ化した．横軸はサイト数$s$を表し，縦軸はページフォールトに要する通信量を$d$の倍数で表している．

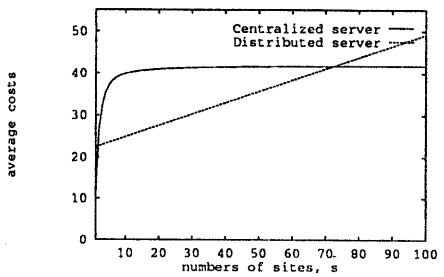


図 11: $r=1$における$C_c$，$C_d$の比較

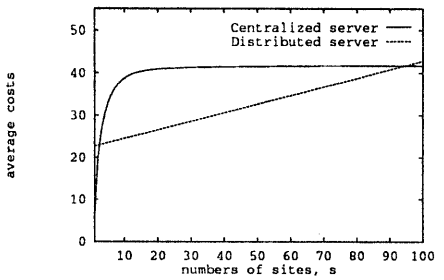


図 12: $r=3$における$C_c$，$C_d$の比較

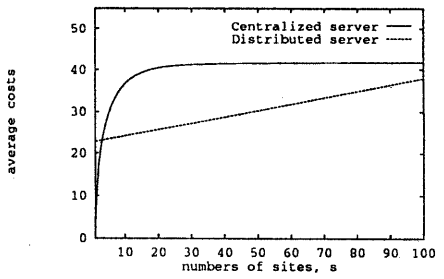


図 13: $r=5$における$C_c$，$C_d$の比較

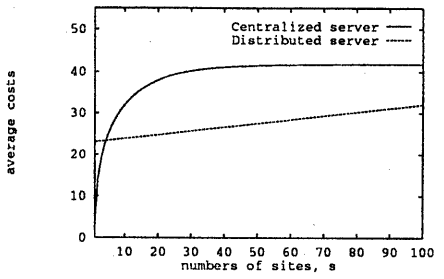


図 14: $r=10$における$C_c$，$C_d$の比較

## 5.4 結果に対する考察

図11～図14から次のようなことがわかる．

**集中型サーバ**

1回のページフォールトで発生するメッセージは，平均して$(2D+2d)$に定まり，サイト数$s$がある値$(s')$以上に増えてもページフォールトに要するコスト$C_c$は変化しない．ただし，クライアントのリード／ライト比が増加すると，$s'$が大きくなり，$C_c$のグラフの立ち上がりは緩やかになる．

**分散型サーバ**

サイト数に比例して，リーダのページを無効にするコストが増大するため，$C_d$のグラフは(クライアントのリード／ライト比$r$によって定まる傾きをもつ)直線になる．ただし，$r$が大きくなると，$C_d$の傾きは小さくなる．

したがって，特に$r$が高い値を示すようなクライアントに対して，分散型サーバは集中型サーバよりも高い性能を示すことがわかる．また，比較的$r$の低いクライアントに対しても，サイト数が一定値以下の場合には，分散型サーバの性能が集中型サーバのそれを上回っていることが，図11からわかる．クライアントの$r$が低く，かつサイト数が多い場合には，ページサイズを小さくして，同じページにアクセスするクライアント数を相対的に減らすことで，集中型サーバよりも高い効率を得ることができる．

## 6 むすび

WAKASHIは，Machオペレーティングシステムの外部ページャとして実現された[BM92]．WAKASHIの提供するDSPHによって，ユーザはプログラム中の揮発データと同様に，ファイル中の永続データを扱うことが可能となった．しかし，以前の実装法では，WAKASHIサーバが集中型サーバとして実現されていたために，サーバのボトルネックやサイト間の不平等など，いくつかの問題があった．本研究で実装した分散型サーバは，これらの問題を解決した．

まず，フォワーディングを実装することによって処理効率が向上することを，理論式を用いた性能評価によって示した．この結果によると，リード／ライト比の高いクライアントに対して，特に分散型サーバが効率的であることが示されている．

ただし，実験に用いた環境ではサイト数の制限があるため，本論文で導出した理論式による予測値を実測結果によって確認することはできなかった．

今後，3サイト以上の環境で実測を行ない，従来の集中型サーバと本研究で実装した分散型サーバの性能を，実測値を用いて比較評価する必要がある．また，DSPHをオブジェクト単位で管理する方式の実現を行ない，両者の実行性能を比較評価することを検討している．

## 参考文献


[BM92] G.Bai and A.Makinouchi: "Implementation and Evaluation of a New Approach to Storage Management for Persistent Data – Towards Virtual-Memory Databases," Proc. of the Second Far-East Workshop on Future Database Systems, Kyoto Japan, Apr.1992, pp.211-220.

[BT92] G.Bai, K.Teramoto, H.Amano, and A.Makinouchi: "WAKASHI, A Distributed Shared Persistent Data Store – It's Implementation and Performance Evaluation –," Technical Report CSCE-92-C11, Dept.Comp.Sci. andComm.Eng., Kyushu Univ., Sept.1992.

[Co90] The Commitee for Advanced DBMS Function: "Third-Generation Database System Manifesto," Memorandom No.UCB/ERL M90/28, Apr.1990.

[FB88] A.Forin, J.Barrera, M.Young, and R.Rashid: "Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach," CMU-CS-88-165, Aug.1988.

[IS92] 乾, 菅原: "分散OS Machがわかる本," 日刊工業新聞社, 1992年10月.

[LH89] K.Li, P.Hudak: "Memory Coherence in Shared Virtual Memory Systems," ACM Trans. on Computer Systems, Vol.7, No.4, Nov.1989, pp.321-359.

[LL91] C.Lamb, G.Landis, et al.: "The ObjectStore Database System," Comm. ACM, Vol.34, No.10, Oct.1991.

[Ma91] 牧之内："オブジェクト指向データベース管理システムのアーキテクチャ," [情報処理], 第32巻, 第5号, 1991年5月.

[Ma92] 牧之内："永続プログラミング言語プロジェクト「出世魚」について," 情報処理学会第44回全国大会, 2H-1, 平成4年3月.

[NL91] B.Nitzberg and V.Lo: "Distributed Shared Memory: A Survey of Issues and Algorithms," Computer, Vol., No., Aug.1991, pp.52-60.

[SZ90] M.Stumm and S.Zhou: "Algorithms Implementing Distributed Shared Memory," Computer, Vol.23, No.5, May 1990, pp.54-64.

[Ud92] 宇田川: "オブジェクト指向データベース入門," ソフト・リサーチ・センター, 1992年8月.