システムソフトウェアと
オペレーティング・システム 70－5
（１９９５．８．２２）

メモリマップされたトランザクションを効率良く実現するための メモリオブジェクトの枠組

國澤亮太　　猪原茂和　　益田隆司

東京大学大学院 理学系研究科 情報科学専攻

Machなどのマイクロカーネルで提供される外部ページャ機能を用いて永続領域のデータをメモリオブジェクトとしてマップすると、クライアントはローカルなアドレス空間にマップされたデータに対して直接に操作ができるので、プログラミング言語で使用するような複雑なデータ構造を扱えるデータベースシステムが実現できる。しかし、現在の外部ページャのインタフェイスではトランザクションのために必要なバージョン化機構が効率的に実現できない。本稿では、トランザクションを効率良く実装するためのメモリオブジェクトの枠組みを提案する。この機構により、カーネル内でシャドウページングをおこないトランザクション間で可能な限りページを共有し、外部ページャがページ単位でなくメモリオブジェクト単位でバージョン化を操作することにより、カーネルと外部ページャの間の通信を減少する

A Framework of Memory Objects for Efficiently Implementing Memory-mapped Transactions

Ryota Kunisawa, Shigekazu Inohara and Takashi Masuda

Department of Information Science, Graduate School of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113

With the use of the mamory mapping method provided by external pagers of todays microkernels, clients can access data in persistent storage directory through memory references. Thus, database system can handle complex data structures. The existing external pagers, however, cannot efficiently support data versioning, which is indispensable when implementing transactions. This paper proposes a new memory object framework for efficiently implementing transactions. With this framework, memory objects are versioned in the kernel so that with shadow paging technique, transactions can share pages as much as possible, and external pagers manage the versioning in the grain of memory objects. Consequently, this framework reduces the communication between the kernel and external pagers.

1 はじめに

大規模なデータベースシステムは、今日のコンピュータアプリケーションには欠かせないシステムである。このようなシステムでは、大量のデータを管理するデータベースサーバがあり、複数のデータベースクライアントが同時に動いてサーバのデータを参照し、更新する。

複数のクライアントが同時に動くためには、データのアクセスに一定の制限を課し、サーバのデータの一貫性が損なわれないようにすることが大切である。データベースシステムでは「トランザクション」の導入により、データのアクセスに制約を課している。

トランザクションの実装には色々な方法があるが、データの構造が複雑で、特にポインタの参照がおこなわれる場合、メモリマップ技術を用いてトランザクションを実装する場合が多い。サーバはクライアントのアドレス空間にデータをマップし、クライアントはローカルなアドレス空間への参照と書き込みでデータベースにアクセスできる。

同一のデータに対するアクセスに一貫性を持たせるために、セマフォやモニタなどでデータに関して排他的アクセスを強いるという古典的な方法があるが、メモリマップされたデータに対するトランザクションに関しては、楽観的な並行性制御がおこなわれることが多い。「楽観的な並行性制御」とは、クライアントそれぞれにデータのバージョンを用意して勝手なアクセスを許し、トランザクションがコミットした時点で、データの一貫性を保ちつつ、データベースに対しての変更をおこなうものである。

メモリマップされたデータベースに対して、楽観的な並行性制御が採用される理由として、以下が挙げられる。

- データの操作は機械語レベルでおこなわれ、履歴をとることが困難である。
- 排他的なアクセスと異なり、複数のクライアントが同時にトランザクションを実行できる可能性が増加する。

これは特に、長い read-only トランザクションと短い read-write トランザクションが共存する分散協調作業環境で有効となる。

また、ページ単位でデータを管理管理することにより、メッセージ通信の要求をハードウェアで検知し、通信の頻度を下げることができる。サーバからクライアントへのデータの供給は、クライアントがページフォルトを起こした時点でなされ、クライアントからサーバへのデータの返還はダーティなページのスワップアウトとして実現される。これは、ページサーバアーキテクチャと呼ばれ、ObServer [1]、O_2 [2]、EXODUS [3]、Object-Store [4] などのシステムがこの方法をとっている。また、楽観的な並行性制御に基づく分散並行性制御もいくつか提案されている (O2PL [5]、DVO [6, 7])。

本稿の構成は以下のとおりである。2 章で既存のマイクロカーネルのインタフェイスを用いて、ページサーバアーキテクチャに基づくメモリマップされたトランザクションを使用し、データベースシステムを実装する際に生じる問題点を述べる。3 章でこれを解決する新しいインタフェイスを提案する。このインタフェイスを使用すると、カーネル内にトランザクション間のバージョン関係を表すメモリオブジェクトを生成する。クライアント間で読み専用として共有するページを持つメモリオブジェクトと、クライアントがそれぞれ書き換えたページを持つメモリオブジェクトをカーネル内で管理することにより、トランザクション間で可能な限り物理ページを共有でき、ページを書き換えた場合に外部ページャへ通知されるメッセージ数を減らすことが可能となる。4 章でその実装方法を述べる。さらに、5 章で新しいインタフェイスの性能評価方法を述べ、6 章でその結果を述べる。

2 問題点

2.1 Mach での仮想記憶の実現

UNIX ならば `mmap()` でファイルからページをマップするが、現在のマイクロカーネルでは、メモリを持つ実体が抽象化されていて、例えば Mach ならば、`vm_map()` で抽象化されたオブジェクト (メモリオブジェクト) からページをマップする。メモリオブジェクトのページングをおこなう外部ページャをユーザが実装することが可能であり、メモリマップされたデータを扱うデータベースを比較的簡単に実装可能である。この場合、外部ページャがデータベースのサーバとなり、そのページャのデー

タをマップするプロセスがデータベースのクライアントとなる。

2.2 Mach で実装した場合の問題点

Mach の外部ページャ機構を使用して、メモリマップしたデータのトランザクションを実装するには、以下の 3 方法がある。

2.2.1 実装方法 1

データをマップするのに使用するメモリオブジェクトは、すべてのクライアントで共有する。

この場合、すべてのクライアントは同じイメージを共有するので、一度に書き込めるクライアントは一つにする必要がある。セマフォやメッセージパッシングを用いて、排他的書き込みを実現すればトランザクションになるが、楽観的な並行性制御とはならないので、本稿の対象とはしない。

2.2.2 実装方法 2

トランザクションそれぞれにメモリオブジェクトを用意する。リードフォルトのページは、サーバのアドレス空間にページを用意し、コピーオンライトによりメモリオブジェクトに供給する。これにより、読まれるだけのページは共有することが可能。ライトフォルトのページは、ページを用意した後、クライアントのアドレス空間に移動させる。問題点は、

- 余計なメッセージが多い

  トランザクションがリードフォルトを起こした場合は、必ずページャに通信が行く。ページャでは、すでにサーバのアドレス空間にあるページをコピー、またはコピーオンライトで張り付けるだけかもしれない。

  現在の Mach のインターフェイスでは、クライアントからリクエストが来る前にサーバがページを供給することは不可能である。

- サーバを書くのが面倒になる

  どのトランザクションがどのバージョンを見ているのかユーザが管理する必要がある。

2.2.3 実装方法 3

データをリードオンリでマップする共有メモリオブジェクトを用意し、書き込まれたページを持つメモリオブジェクトはトランザクションごとに用意する。クライアント毎のリードフォルトが通知されることはないが、ライトフォルトが起こった場合、ページを自分専用のメモリオブジェクトからマップし直して、クライアントはリトライする必要がある。この方法の問題点は、

- ページフォルト中にページのマップをし直すことが、現在の Mach のインターフェイスで実現可能かどうか不明。可能であるとしても、ユーザがマップのためのシステムコールを発行する必要がある。

- データへのアクセスは CPU のインストラクションレベルであり、これをリトライするのは困難。

「実装方法 2」、「実装方法 3」のいずれの場合も、データベースにコミットされたバージョンと、各トランザクションが更新中の一時的なバージョンの管理をおこなう必要があるが、既存のカーネルを用いた場合、バージョン管理はユーザが設定するページャの役目になる。

3 解決法

トランザクション間の親子関係

データベースのサーバでは、トランザクション間のデータのバージョンを管理する必要がある。データベースの永続領域に存在するデータは、コミットされたトランザクションによって作成される永続的なバージョンであり、実行中のトランザクションが必要とするデータは、コミットされたバージョンと、自分が書き換えてしまったデータを表す一時的なバージョンである。トランザクションがデータを必要とする過程は、コピーオンライトの手法でマップされたページの処理と全く同じである。図 1 において、永続領域にある (コミットされた) バージョン A に対してトランザクション X が始まった場合、X は A のページを参照し、データを書き込んだ場合にページをコピーして所有する。

別のトランザクション Y が実行を始めると、Y も A のデータをコピーオンライトで参照する。X がコミットした場合、データベースには新しくコミットされたバージョン B が作成される。それ以後に始まるトランザクション Z は、B をコピー

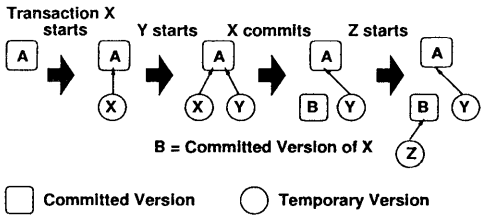

図 1: バージョン間の親子関係

オンライトで参照する。Y がコミット可能であるか、また Y は X がコミットした影響を受けて B を参照するようになるかはデータベースの実装による。Oberon System [8] 等のオブジェクト指向 OS では、ユーザが独自に基底クラスの変更をおこない、システムの挙動を変更することができるが、その変更はそのユーザにのみ有効化することができる。これは、クライアントがそれぞれトランザクションをコミットして独自のバージョンを作成することを意味している。

ここで、コミットされたバージョンも、一時的なバージョンも、それぞれのバージョンを持つメモリオブジェクトとして考えることが可能である。現在の外部ページャのインターフェイスでは、コピーオンライトされたページが属するメモリオブジェクトは変更されないが、ページに書き込みが起こり、コピーされた時点で別のメモリオブジェクトの所有物とすれば、前述のようなバージョン管理が可能である。本稿で提案する新しいインタフェイスでは、親メモリオブジェクトと子供メモリオブジェクトを作成し、カーネル内でリンクを張る (図 2) ことによりページのバージョン管理をおこなう。ページのマッピングは子供メモリオブジェクトに作成される。この新しいメモリオブジェクトを、「トランザクショナルメモリオブジェクト」と名付ける。

フォルト時の処理は以下のようにおこなわれる。

リードフォルトの場合 (図 3)

1. フォルトがカーネルに通知される。
2. 子供オブジェクトにページがなく、今までに書き込んだこともなければ、フォルトは親オブジェクトに通知される。
3. 親オブジェクトがページをリクエストする

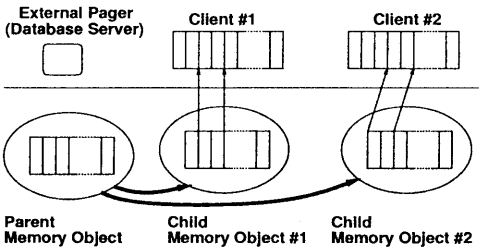

図 2: トランザクショナルメモリオブジェクト

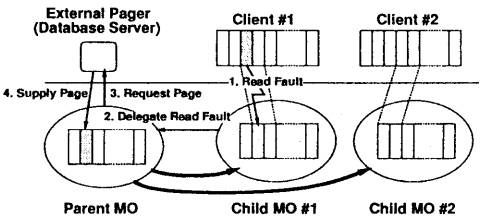

図 3: リードフォルトの処理

4. ページは読み専用でマップし、親メモリオブジェクトで共有する。

もしクライアントが書き込んだことのあるページならば、フォルトは親オブジェクトに通知されず、ページのリクエストは直接子供オブジェクトから来る。

別のクライアントが同じページでフォルトを起こした場合、ページがすでに親オブジェクトにあれば、データのリクエストは外部ページャに通知されないので、無駄なアップコールが減少する。

ライトフォルトの場合 (図 4)

1. 1 から 4 まではリードフォルトの場合と同じ
5. カーネル内で子供オブジェクトにページをコピーする。

リードフォルトの時と同じく、一度書き込んだことのあるページならば、子供オブジェクトが直接ページをリクエストする。

この方式でおこなえば、誰かがライトフォルトを起こしたページでも親オブジェクトに溜るので、以後のリードフォルト、ライトフォルトでのアップコールの数が減少する可能性があるが、親オブジェ

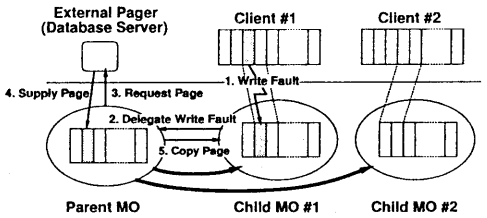

図 4: ライトフォルトの処理 (1)

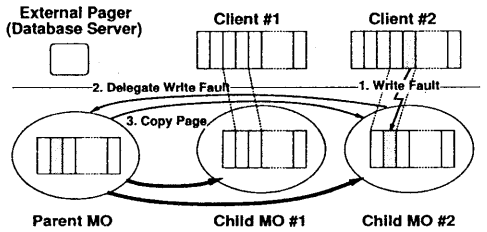

図 5: ライトフォルトの処理 (2)

クトのための物理ページと子供オブジェクトのための物理ページができる。3. 4. 5. のシークエンスの代わりに、親オブジェクトにページがなかった時点で、直接子供オブジェクトからページのリクエストをおこなえば、子供オブジェクトのための物理ページのみがでるので、どちらが効率の良いものであるかは疑問の余地があるかもしれない。

リードフォルトの場合と同じく、他のクライアントがリード(またはライト)フォルトを起こしたページであっても、外部ページャにページのリクエストは起きない(図 5)。

以上の考えは、前節の「実装方法 3」を効率良く実現するものである。前節の「実装方法 2」に関する解決方法として、あるクライアントがページをリクエストした時点で、そのデータをマップしている他のクライアントにもページを供給することが考えられる。(ページのプリフェッチ、キャッシュのインジェクションと等しい。) コピーオンライトで他のクライアントにページを供給しても、クライアントがそのページに書き込みをおこなうまでは物理ページのコピーは起こらないので、余分なアップコールが減少し、また読み専用の物理ページも共有できる。しかし、ページを実際に参照しないトランザクションに対してもページを供給すれば、そのためのシステムコールは無駄になってしまう。

4 実装

本稿で提案するトランザクショナルメモリオブジェクトを、マイクロカーネルである Mach V3.0 (MK83) 上に実装した。その主な変更点は以下の通りである。

- システムコール `vm_map_copy()` を追加

  既存のシステムコールである `vm_map()` と同じく、メモリオブジェクトのページを自分のアドレス空間にマップするが、この際に親メモリオブジェクトと子メモリオブジェクトのポートを指定できる。この 2 つのメモリオブジェクトは、カーネル内でポインタによりリンクされる。

- カーネル内手続き `vm_fault()` の変更
- カーネル内手続き `vm_fault_page()` の変更

  「3 解決法」で述べたことをおこなう。

- カーネル内手続き `vm_pageout_page()` の変更

  ページのエントリを完全に消してしまうと、子供メモリオブジェクトのページが変更されたかそうでないか区別がつかなくなってしまい、それ以後のページフォルトで親メモリオブジェクトと子供メモリオブジェクトのどちらにページをリクエストすればいいのか分からなくなってしまう。これを避けるために、子供メモリオブジェクトでダーティなページをページアウトする場合、フェイクエントリを残し、それ以後のフォルトは子供メモリオブジェクトで処理するようにする。

- カーネル内手続き `vm_page_free()` の変更
- カーネル内手続き `memory_object_data_supply()` の変更

  フェイクエントリを扱えるようにする。

- トランザクショナルメモリオブジェクトを生成、削除、コピーするカーネル内手続きの作成。

  既存のメモリオブジェクト用手続き (`vm_object.c`) を参考に作成。

なお、メモリオブジェクト間のリンクに使用するポインタには、コピーオンライトに用いるポインタ

を利用することにより、データ構造体のサイズの変更は起きない。

ソースに追加した行数は、仮想メモリ関係のコード約21700行に対して、約1700行となった。また、DEC Station 5000 (MIPS R3000) のマシンでコンパイルしたところ、サイズは 1072412 バイトから 1082640 バイトとなった。

追加した行数のうち、約1000行は既存の関数に新しい引数を加えてほぼそのままコピーしたものであり、新しいインタフェイスを考慮して既存の関数を変更すると、追加する行数はより少なくなり、最終的なカーネルのサイズもより小さくなることが予想される。

5 性能評価手法

性能評価には OO7 Benchmark [9] を使用した。OO7 Benchmark は C++ のデータ構造を永続領域に置くオブジェクト指向データベースの性能を測定するためのベンチマークであり、CAD/CAM/CASE アプリケーションが用いるデータ構造をデータベースにしている。

オリジナルの論文では、異なるデータベースで性能の違いを調べるために、一つのクライアントがおこなうトランザクションの実行時間を測定していた。

今回は、複数のクライアントを同時に走らせ、新しいインタフェイスを用いて実装されたサーバと、既存のインタフェイスでで実装されたサーバで、クライアントの実行時間がどう異なるか測定する。

5.1 OO7 Benchmark

OO7 Benchmark で用いるデータ構造は、数種類の型のオブジェクトがポインタでつながった構造をしている。

OO7 Benchmark には Traverse と Query があるが、Traverse のみを実行した。Query を実現するには、普通のプログラミング言語にはない bulk type (set, multiset, list) 等のデータ型が必要であり (C++ のテンプレート機構では実現可能)、また Query の結果がデータベース上に作成される場合は、動的な要因により解析が難しくなる。

今回使用した OO7 Benchmark (Traversal) の特徴

- データベースの構造は、一度作ったら変更されない。

  ベンチマークの途中でオブジェクトを生成しない。また、ポインタ参照がなくなったり、オブジェクトを破壊したりすることがない。つまり、トランザクションの途中でガベージコレクタ、NEW()、DISPOSE() 手続きが起動されることがない。

- データベースのデータを読んでからすぐに更新する。

  例えば、Traversal No.2 では、トラバースした先のオブジェクトの 2 つの属性値 (整数) を交換するが、交換するためには、はじめに属性値を読む必要がある。また、Traversal No.3 では、属性値が偶数ならば 1 減し、奇数ならば 1 増やす。いきなりライトフォルトが起こる確率は 0 に等しい。

データベースのクライアントのプログラムは、データをマップするシステムコールが異なる以外は、既存のインタフェイスを用いた場合と新しいインタフェイスを用いた場合ではまったく同じである。サーバのプログラムに関しては、既存のインタフェイスを用いた場合、データのバージョン間の親子関係を把握するために、子供のバージョンから親のバージョンに向けてポインタを張る必要がある。また、子供のバージョンを表すメモリオブジェクトはポート ID としてカーネルから通知されるので、ポート ID からバージョンを調べるためのデータ構造と関数を必要とする。

6 結果

ベンチマークには、DEC Station 5000 (MIPS R3000, 25MHz) を使用した。メモリは 12MB 実装されていて、カーネルが読み込まれた後のユーザが使用可能なメモリは 9.3MB 、1 ページは 4Kb なので、約 2380 ページである。

データベースのデータは、OO7 Benchmark で small とされる大きさで作成し、Composite Part 当りの Atomic Part の数は 3 とした。この場合、データは全部で 848 ページとなる。ベンチマークは Traverse 2a を使用した。クライアントがアクセスするのは 766 ページ、書き換えるのは 498 ページである。

クライアントのアクセスパターンを表 1に示す。

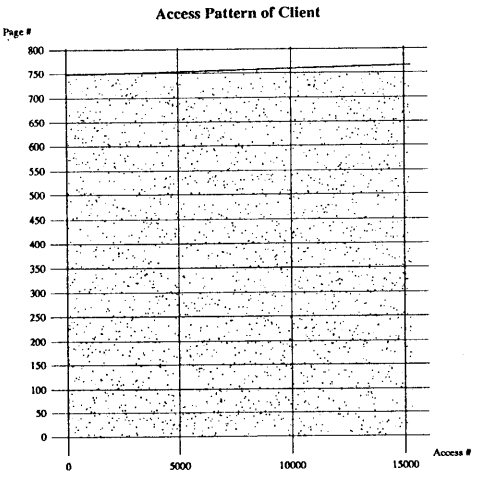

表 1: クライアントのアクセスパターン

横軸はデータベースへの時間であり、縦軸はアクセスするページ番号を表す。アクセスはランダムであるが、ルートセットである特定のページ範囲には頻繁にアクセスしていることが分かる。

前述のとおり、OO7 Benchmark は複数のクライアントを同時に実行した場合の評価結果を求めるものではない。本稿では、同時に実行するクライアントは同じアクセスパターンを持つので、ページが共有される確率が極めて高いことが予想される。

既存のカーネルを用いた場合の実行結果を表 2に示す。

同時に実行したクライアント数	1	2	3
実行時間	71	207	480
メッセージ数	1469	2640	3454
ページリクエスト数	901	1618	2114
read() 数	835	905	1095
write() 数	566	1020	1338
無駄なメッセージ数	66	713	1019

表 2: 既存のインタフェイスを用いた場合

表中の数字は、クライアント毎の数字を表す。ここで、「無駄なメッセージ数」とはクライアントでフォルトが起きて、親のバージョンのページが必要となったが、そのページはすでにサーバの物理ページとして存在していて、本来ならばページの read() をする必要がなかった場合に、無駄なアップコールとなったページリクエストの数である。つまり、

「ページリクエスト数 = read() 数+無駄なメッセージ数」

という関係がある。表中の数字で等号が成立しないのは、数回おこなった結果の平均を表示しているからである。「write() 数」に関しては、トランザクションがコミットしてから連続的に戻ってくるダーティなページも含んでいるので、トランザクションの途中でページャによって書き出されるページの数を表しているわけではない。

トランザクショナルメモリオブジェクトを用いた場合の実行結果を表 3に示す。

同時に実行したクライアント数	1	2	3
実行時間 (秒)	71	108	388
メッセージ数	1419	1440	2456
ページリクエスト数	41	259	789
read() 数	41	259	789
write() 数	539	757	1287
親バージョンでのページリクエスト数 (親バージョンでの read() 数)	767	767	767

表 3: 新しいインタフェイスを用いた場合

「read()」数は、クライアントが書き換えたページがページアウトされたため、親メモリオブジェクトでなく子メモリオブジェクトからページリクエストが来たことを表す。無駄なアップコールがないため、

「ページリクエスト数 = read() 数」

が成立する。また、

「write() 数 = read() 数+498(書き換えたページの数)」

が成立する。子供メモリオブジェクトでフォルトが起き、親メモリオブジェクトからページリクエストが来た場合、これをクライアント毎の値として表示することはできない。そこで、「親バージョンのページリクエスト数」は、すべてのクライアントがトランザクションを終えるまでに親メモリオブジェクトから来たページリクエストの数を示す。この値は、親バージョンでのページ read() 数に等しい。

一度の無駄なアップコールに必要な時間は、「外部ページャがすでにページを持っていた場合のアップコール時間 (1.27msec) ＋ページをアロケートする時間 (0.0541msec) ＋ページをコピーするのに必要な時間 (1.92msec) ＋外部ページャの実行時間」である。既存のインタフェイスを用いた場合の実行時間から無駄なアップコール数に見合うだけの時間を引いた時間は、新しいインタフェイスを使用した場合の実行時間に加えて、大幅に大きいことが分かる。これは、無駄なアップコールをユーザのプロセスで処理するので、そのために必要なユーザのページがページングの対象になっているからだと考えられる。また、既存のインタフェイスを用いた場合、バージョン間をたどるにはメモリオブジェクトの ID からトランザクションを表す構造体を調べ、その構造体を参照する必要があるが、カーネル内でメモリオブジェクトは構造体で表されるので、新しいインタフェイスを用いた場合、バージョン間をたどるのはポインタの示す先を参照することで可能となる。

新しいインタフェイスを用いた結果を観察すると、親メモリオブジェクトのページは完全にキャッシュされてしまっていることが分かる。新しいインタフェイスによる利点が強調されてしまったベンチマークとなってしまった。

7 おわりに

本稿では、既存のマイクロカーネルのインタフェイスを用いてデータをメモリにマップしたトランザクションを実装する場合の問題点を調べ、トランザクションで作成されるデータのバージョン間の関係の情報をカーネル内に供給することにより、トランザクションがより効率的に処理されることを示した。また、カーネル内で使用されるデータ構造を再利用することにより、カーネルに加える変更は最小限となり、既存のコードもそのまま利用可能である。

複数のトランザクションを同時に実行する場合の、より現実的な評価方法を確立することが今後の課題である。

参考文献

[1] M. Hornick and S. Zdonik, "A Shared, Segmented Memory System for an Object-Oriented Database", *ACM Transactions on Office Information Systems*, Vol.6, No.1 pp.70–95 (1987)

[2] Françious Bancilhon, Claude Delobel and Paris Kanellakis, *Building an Object-Oriented Database System — The Story of O_2*, Morgan Kaufmann Publishers, Inc., San Mateo, CA (1992)

[3] Michael J. Carey and David J. DeWitt, "The Architecture of the EXODUS Extensible DBMS", *Proceedings of the International Workshop on Object-Oriented Database Systems*, pp. 52–65, IEEE, Pacific Grove, CA (September 1986)

[4] C. W. Lamb, Gordon Landis, J. A. Orenstein and D. L. Weinreb, "The Object Store Database System", *Communications of the ACM*, Vol.34, No.10 pp.50–63 (October 1991)

[5] M. J. Franklin and M. J. Caray, "Client-Server Caching Revisited", *Distributed Object Management*, pp. 57-78, Morgan Kaufmann Publishers, Inc., San Meteo, CA (1994), paper from International Workshop on Distributed Object Management (IWDOM)

[6] S. Inohara, Y. Shigehata, K. Uehara, H. Miyazawa, K. Yamamoto and T.Masuda, "Page-Based Optimistic Concurrency Control for Memory-Mapped Persistent Object Systems", *Proc. of the 28th IEEE Hawaii Int'l Conf. on System Sciences (HICSS-28)*, Vol. II, pp.645–654, (January 1995)

[7] M.Hara, K. Yamamoto, K. Uehara, H. Miyazawa, S. Inohara and T.Masuda, "Memory-map Based Transaction System for Managing Persistent Objects", SWoPP 別府'95 プログラミング分科会 (PRO-5) で発表予定, (August 1995)

[8] N. Wirth and J. Gutknecht, *The Oberon System*, Computer Science Report 88, Departement Informatik, ETH Zürich (1988)

[9] Michael J. Carey, David J. DeWitt and Jeffrey F. Naughton, "The OO7 Benchmark", *SIGMOD Rec. (USA)*, Vol.22, No.2 pp.22–31 (June 1993)