データベース・システム 83－5
（1991. 5. 24）

永続的プログラミング言語における
オブジェクト識別子の主記憶内表現について

鈴木 慎司　喜連川 優　高木 幹雄
(東京大学 生産技術研究所)

Persistent Programming Language や Object Oriented Database においてアプリケーションの実行速度を向上されるためには、データベース中のオブジェクトアクセスを高速化する必要がある。とくにオブジェクトを主記憶中にキャッシュし、できるだけハードウェアの参照メカニズムに近いかたちで参照を解決することが重要である。本稿ではそのような参照の効率的なキャッシング（参照形式の変換と捉えることもできる）を行なうためのオブジェクト識別子の表現形式について議論をするとともに、主記憶管理の重要性と問題点に触れる。

Possible In-memory Representations of Object Identity for implementing a Persistent Programming Language

Shinji Suzuki, Masaru Kitsuregawa, Mikio Takgai
Insitute of Industrial Science, University of TOKYO
7-22-1 Roppongi Minato-ku Tokyo, 106
suzuki@tkl.iis.u-tokyo.ac.jp

It is quite important to optimize access time for objects in database when we try to execute applications as fast as possible. Especially caching objects in main memory and to refer the cached objects in a way which is natural to underlining hardware is essential, maybe by converting the form of reference. This note discusses about the possible representations of object identity,reference,in memory in addtion to issues in main-memory management to achieve that goal.

1 はじめに

７０年代に提案されたリレーショナルデータベース (RDB) は、長い間にわたる研究やインプリメンテーションの改良によって市場でも受け入れられるようになったが、それはほとんどがビジネスアプリケーションと呼ばれるテーブルの形で自然に表現できる単純なデータを対象とする応用においてである。 エンジニアリング応用特に cax と総称される cad/cam, case 等の分野では、ほとんど RDB が利用されることなく cax ソフトウェアのベンダはファイルシステムを基に簡易的なデータベースを作成しアプリケーションを構成してきた。（といわれている）

その原因としては、それらのアプリケーションが必要とするデータ表現能力と RDB が提供するモデルの間のミスマッチから生じるプログラミングの困難さ、そして RDB の使用による性能低下があげられている。

最初の問題の解決策としては、アプリケーションを記述するのに十分な表現力とモデル化の能力を持った言語を備えたデータベースシステム（ＯＯＤＢ？）を用いる、あるいはコンパイラや実行時ライブラリのサポートを加えてプログラマに負担をかけずにプログラミング言語とデータベースの間で型の変換を行なう（Persistent Programming Language を使用する）方法が考えられる。

いずれの方法を用いたとしても特殊なハードウェアやオペレーティングシステムのアーキテクチャに依存しない限り、主記憶上のデータと二次記憶上のデータへのアクセス速度には格差が生じる。この隔たりを十分に小さく押えられない限り CAX アプリケーションは、プログラマへの負担にもかかわらずファイルベースで開発がすすめられることになるだろう。

本稿では通常行なわれる RDB の使用法が cax のアプリケーションに性能上悪影響を与える原因の一つについて考察した後、その解決となりうるデータ（以後、プログラミング言語の型を持ったデータという意味でオブジェクトと呼ぶ）アクセスの方法について言及する。次にそのアクセス方式（キャッシュ方式とも考えられる）を採った場合に、どのような形で主記憶中のオブジェクト識別子を表現するのが効率的であるかを, 実験データとともに考察する。最後に、仮想空間の管理（キャッシュの管理とも考えられる）のうち空間の回収を中心に考慮すべき点をいくつか上げる。

2 参照のキャッシュ

ＲＤＢを利用したアプリケーションの性能を考える時、join の高い負荷がよく問題にされる。例えば、図１のようなリレーションがあった場合に brian の車の色を調べるには”品川ひ-1234”というキーで２つのリレーションを join（正確には、このキーでリレーションＢに対してセレクションをかけるというべきかもしれないが）する必要があり、この処理がアプリケーションの実行性に悪影響を与えるといわれることがある。[SMIT87]

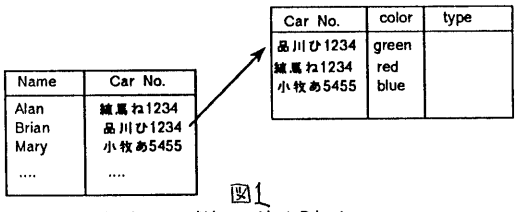


図1

A reference within an object Brian to a car.

確かにこの車の参照がナンバというキーによってではなく、絶対的なアドレスで参照されていれば、セレクションをする必要もなくなるわけであるが、データベース上のデータのように頻繁に更新、削除され、その結果データの移動や空き領域の管理が必要となるような対象を扱う時には、常に絶対アドレスの参照を有効に保つコストが参照のコストの減少分よりも大きくなることが多いであろう。そして間接参照を用いたほうが利用者の簡便性がはかられると思われる。ファイルシステムに例をとると、ファイル名というキーによる間接参照を利用することでディスクのブロック管理を簡単かつ効率的にし、利用者の便宜を図っていると考えることができよう。

そのように考えてみると、むしろ問題点は間接的な参照が存在することではなくその間接参照を無駄に何度も辿ることであると思われる。一度辿られた参照は、次に参照が行なわれる時にも、同じアドレスに行きあたることが多いはずである。（とくに２つの参照の間隔が短いほど、その確立が高い。）こうした間接参照の柔軟性と参照結果のキャッシングによる性能低下の回避は、MMU のページテーブル（あるいはセグメントテーブル）と TLB の組合せなどにも見られる。

ところが、RDB を用いてプログラミングを行なう場合、この参照結果のキャッシングが考えられることは少なかったようである。一つに

は、これまでRDBが主に使用されてきたアプリケーションにおいては、ディスク上のデータを検索することが最大の目的であり、データが主記憶に読み込まれたあとの計算は全体の処理時間からみるとごくわずかであることがあげられる。しかし、これはCAXアプリケーションにおいては成立しない。従ってRDBを先進的なアプリケーションにおいて使用するためには参照の効率的なキャッシングが必要になると思われるのであるが、これまでのように通常の言語にqueryを埋め込むという形式のプログラミングでは、参照に関するキャッシュの管理のみでなく、型の変換やオブジェクトを格納するアプリケーションプロセスの仮想メモリの管理をプログラマが行なわなくてはならず現実的ではない。

そこで有用であると考えられる方法はプログラミング言語に永続性を加え、永続的なオブジェクト（プログラミング言語の型システムに従う）の扱いを可能とした上でコンパイラや実行時ライブラリによって自動的な参照のキャッシングを行なうことである。その際に型システムの相違を吸収するような変換を行なうことにより、storage subsystemとしてRDBを使用することも可能であろう。[3rdGDBM]でもこのようなアプローチの有効性について述べている。次章ではこの参照のキャッシングを効率的にする（キャッシュヒットの検査のためのコストを押える）ために、どのような形式のオブジェクト識別子を利用するのが好ましいかを考察する。この問題はオブジェクト指向データベースの実装を行なう時にも避けて通ることができないと考えられる。既に述べたように記憶領域の柔軟な管理のためには間接参照が有用であるし、実行時の性能の面からはできるだけ直接的な参照を行ないたいからである。

3 絶対アドレス or 間接参照？

間接参照

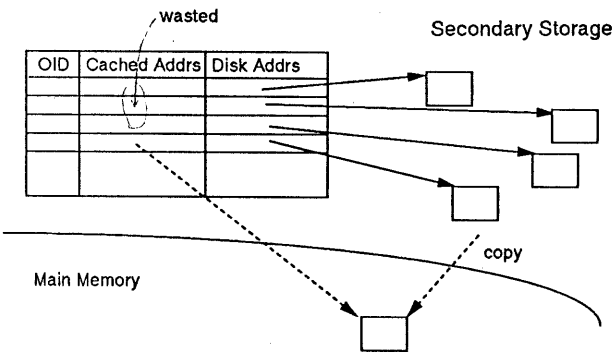
前章で述べた車の色の取り出しの例をもとに、もっとも単純な参照のキャッシュ方法を考えてみると、次の図2によって表現できるであろう。今回は主記憶中でのアクセスコストを考えるので二次記憶上のoid表現については深く考慮しない。ここでは、二次記憶上のoidは論理的なものでオブジェクトそのものとは別にoidからディスクアドレスへの変換表が存在するとする。図2の例はこのような二次記憶上のオブジェクトテーブルにオブジェクトの仮想空間内でのアドレスを記憶する領域を用意しておくことを示している。

この方法はシンプルでわかりやすいのであるが、次のような重大な欠点を持っている。

1 オブジェクトテーブルのサイズの肥大化にともなう記憶領域の無駄とテーブル参照のためのI/O増加の可能性。

2 常に全オブジェクトを含むテーブルの検索を行なうのでエントリ検索時のプローブ回数が増化。

3 自然な形の実装を考えた場合、このテーブルの管理はデータベースのキャッシュマネージャで行なわれるため、アプリケーションプロセスが参照を解決するためには比較的コストの高いプロセス間通信を行なわなくてはならない。



(1),(2)を解決する方法としてROT(resident object table)を使用する方法がある。これは主記憶内に存在するオブジェクトのoidとそのアドレスの対応を管理するテーブルである。必要なオブジェクトしかこのテーブルに加えられないので、アプリケーションの持つ参照の局所性にもとづき(1),(2)の問題を回避することができる。またこのテーブルをアプリケーションプロセス内に置くことでプロセス間通信が生じる問題も解消することができる。

このROTを用いた方法は、必要に応じてオブジェクトの読み出しと、追いだしが行なわれROTが書き換えられるという点を除いてsmallta 80に見られるoop (object-oriented pointer)と同じであり、smalltalk-80にオブジェクトレベルの仮想記憶を付加したLOOM(large object-oriented memory) [bhwa]において採用された方法である。(orionも同様)

この方法の問題点としては、オブジェクトアクセスの度にROTのpresence bitの検査及びアドレスの取り出しという余計なメモリ参照が入ることである。(特殊なポインタの値を用いることでpresence bitの代替をすることも可能)

もう一つの問題はC言語のポインタとROTを用いた参照の実現の間に意味的なギャップが存在することである。C言語のようにポインタがオブジェクトの中間を指すをことを許すとポインタとオブジェクトアイデンティティを同一視することができなくなる。従って<oid,adress>の変換表はオブジェクト1個に対してではなく、ポインター一つについて一つ用意する必要がある。以下のコードについて考えてみる。

```
int *a = \&int_array[0];
int *b;
b = a;
++a;
```

この時、aがインクリメントされた後もbは配列の先頭を指していなくてはならないが、ROTのエントリを共有した場合にはbの参照先もインクリメントされてしまう。これを防ぐためにひとつひとつのポインタにROTのエントリを割り当てることにすると、動的にヒープ上に割り当てられたオブジェクトの型をも確定する必要が生じる。それはそのオブジェクト中のポインタに対して新たにROTエントリの割り当てを行なわなくてはならないからである。そのような制限が生じてしまうと既存のコードが利用できなくなってしまう。

この2つの問題を解決する方法として、ポインタ表現を<oid>+offsetの形式で表現する方法が考えられるが、ポインタ長が長くなり実行速度を犠牲になるし、OS管理領域やファイルシステム中の（ポインタ表現がことなる）データのアクセスができなくなってしまう。ただこの表現方法を用いればアレイアクセスの境界チェックも可能になり、<oid>+offsetの計算の頻度をうまく抑えることが可能で、ポインタ表現の差異が問題とならない時には興味深い方法である。

一方ROTを用いる利点としては、全ての参照がROTを介して行なわれるため主記憶が不足した場合にオブジェクトを主記憶から追い出す(stashing)時にもdangling referenceが生じない、compacting garbage collectionをする場合にオブジェクトの移動が容易に可能であるし、bakerのアルゴリズムに従ってincremental なcompacting gcをする場合にもフォワーディングポインタを置く必要がない、といったことがあげられる。

絶対アドレス参照

前節では、間接参照について述べたのでここでは絶対アドレスを用いた参照について考える。基本はps-algolの実装において採用されたフラグビットによってポインタの種類を識別する方法である。[PS-ALGOL] ポインタの最上位ビットが1の時にはその参照が二次記憶上のオブジェクト識別子を含み、0の時には有効なメモリアドレスを含むように制御されている。これにより、永続性を付与したことによるオーバヘッドは、オブジェクトアクセス時にポインタのタグを調べることだけになる。以後では主記憶アドレスを含むポインタを有効なポインタと呼ぶことにし、有効でないポインタを検出し、参照されているオブジェクトを主記憶中に読み込んでポインタの値を読み込みアドレスで書き換える動作をオブジェクトフォールティングと呼ぶことにする。（pointer swizzlingとも呼ばれる）

当研究室で開発しているP3LというPersistent Programming Languageの実装では、以下に述べるように、この方式を次のような点で改良している。

1. PS-AlgolのROTのアクセスが不要になり、ポインタのタグ検査のみで済むこと はオブジェクトのアクセスコストを大きく軽減するが、それでも1度のオブ ジェクトアクセスごとにタグチェックを行なう負荷が大きい。またポインタが 別の関数に渡された時には、無駄なオブジェクトフォールトを検出してしまう。

2. 参照できるdbのサイズが2**(ポインタのビット長-1)に限定されてしまう。この問題にはLOOMと同様に二次記憶上の識別子の長さと、主記憶上の識別子（アドレス）の長さを替えることで対処している。

最初の問題をもう少し詳しく説明する。これは次のような場合に起こる。struct st {int x,y} *p; が二次記憶上のオブジェクト識別子を含んでいるとする。このとき

```
foo( struct st *p )
{ return p->x; }

main(){
  cout << foo(p);
  cout << p->y;
}
```

というプログラムを実行すると関数fooの中と、mainの中の2カ所でオブジェクトフォールトを検出する。ただし2度目以降ではオブジェクトの読み込みは行なう必要はないので最初の時よりもフォールティングの処理に要する時間は少なくて済むが、メモリのアクセス速度とくらべるとかなりのオーバヘッドとなる。アクセスの度の検査、複数回のフォールト検出の問題を

解決するためには、ポインタの値をできるだけ早いうちに検査するとともに検査回数を減らすことが重要になる。

P3Lではポインタの値が使用される時でなく、読み出されるときに検査をすることでこの目的を達している。次のデータ構造を考えてみる。

```
struct anImage {
  char  title[32];
  int   vwidth, hwidth;
  pixel * image;
// pixel==unsigned char*
};
```

このようなデータ構造へのポインタpを引数とし、その画像の平均階調を 計算する関数は、

```
void mean(struct anImage *p)
{ pixel *pix = p->image;
  int size = p->vwidth * p->hwidth;
  int  s=0, i;
  for(i=size;i;--i) s += *pix++;
  return s / size;
}
```

と書ける。PS-Algolの方法では$p \rightarrow image, p \rightarrow vwidth, p \rightarrow hwidth, *pix$というポインタ(p)の使用時にポインタの検査が行なわれるのに対し、P3Lでは、$p \rightarrow image$というポインタ($p \rightarrow image$)の読みだし時に検査を行なう。一般にポインタの指すオブジェクトが参照される回数はポインタの値が読み出される回数よりもずっと多いためこの改良は有効である。ただわずかな例外も存在する。ポインタのテーブルを用いて集合を表現したときに、空きスロットを捜すためにテーブル中のポインタをnullと比べながらスロットをプローブすることがある。この時はポインタの読みだしのみしかおこなわれず、不必要なフォールティングが生じる。この問題は空きスロットの管理のためにフラグビットを設けるとか、空きスロットをリストで繋ぐといった方法で回避できる。

この絶対アドレスを利用したオブジェクト識別子(既に述べたようにCの場合あくまでポインタであって識別子ではないが)の表現の利点と欠点はROTを利用したものと対照的になる。すなわち、オブジェクトアクセスは高速で、C言語のポインタとの相性は良いが、compacting gcやstashingを行なうことが難しくなる。

この2つの方式のいずれを採用すべきであろうか?この問題には容易に答を与えられそうにない。アプリケーションが何を要求するかによって全く評価基準が変わるからである。普遍的な回答を導き出そうとするならば、各々の方式の利点や欠点がどの程度のものであるかを正しく把握する必要があろう。次章ではプログラムの実行速度という観点から両者を比較した結果について述べる。

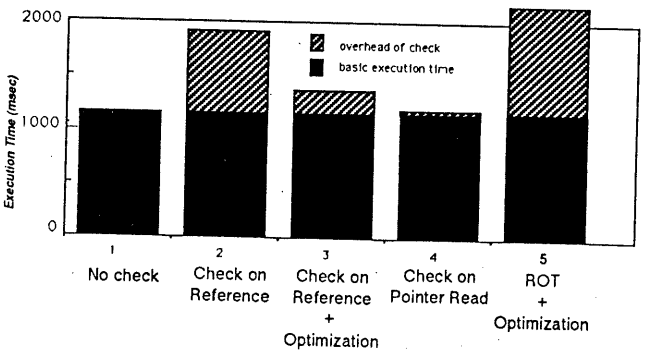
4 オブジェクト識別子(参照方式)の比較

この章ではこれまで上げた方式、特にPS-Algolの方式とP3Lで用いた方法、をプログラムの実行速度という点で比較した実験結果をを示す。

対象としたアプリケーションは、指定したテキストファイル中の単語の出現回数を数えるプログラムである。出現回数を効率的にカウントするためにプログラムは<単語, 出現回数>のペアの集合を単語についてハッシュしテーブルに格納している。衝突の解決にはテーブルを前方にプローブする方法(open hashing)を用い、チェイニングによる解決ではない。このことはポインタによるリンクのトラバースが少なくなるためにP3Lの方式に有利に働く。一見アンフェアに見えるが配列のアドレス計算をもちいてチェインをたどるコストを抑えるテクニックはよく知られたものであることを考えれば、それほど恣意的な例題でもない。

ハッシュテーブルのサイズは2048個とし、5729ワードの単語を含んだファイルを入力とした。ここで問題としているのはプログラム実行時のCPUコストであるのでディスクI/Oやコンソールへの出力時間は計測しない。プログラムは300行程度で関数呼び出しのネスティングも浅く、実行のパターンも一定しているので典型的なアプリケーションとはいえないが、識別子の表現形式を選択する上で参考となろう。

プログラムはC++で記述されSymmetry/S81の上でg++コンパイラによってコンパイルされた。時間の計測はSymmetry内部のカウンタを使って測定した。その結果を以下に示す。



Evalution result of various Access Mechanism

(1) は何の検査もしない場合の実行時間を示している。勿論別のアルゴリズムを用いてプログラムを書き替えたり、レジスタ変数の利用によって、より速いプログラムをつくることはできるが、今回の比較においては理論的な下限値と考えることができる。

(2) は PS-Algol の方式のポインタ検査をした場合の実行時間である。検査はインライン関数によって行なわれているため、関数呼び出しのオーバヘッドはない。

(3) は PS-Algol の方式のポインタ検査に共通式除去のような最適化を行なった場合の実行時間である。この最適化は、コンパイラがポインタの値が有効であることを静的に保証できる場合には、検査のコードを除去することで行なわれる。例をあげると struct st { int x, y, z } *p; というポインタがあり、

```
p->x;
p->y;
p->z;
```

という連続的なコードがあった時には、最初の x のアクセス時に p を検査すれば y と z のアクセス時には検査する必要がない。このような場合には検査コードを挿入しない。ただし、

```
p->x;
foo(p);
p->y;
p->z;
```

のように間に関数呼びだしが入った場合には、このような最適化は行なえない。foo の実行によって p がリロケートされたり、stash されたりするかもしれないからである。もっともより進んだコードの解析を行なえば関数呼びだし後も p の有効性を保証できる場合も考えられる。たとえば foo の中にポインタ参照が存在しなかったり、p の参照のみであった場合などが考えられる。

(4) は P3L でもちいた方法であり、コンパイラの行なう最適化は一切考慮せずポインタの値が読み出される全ての箇所に検査のためのコードが挿入されている。

最後の (5) は ROT を用いた場合を想定した。ポインタのアクセス時には関数コールを必ず行ない、その関数内でビットシフトと排他論理和を用いたハッシュ計算とメモリアクセスを一度行なってリターンするようになっている。コリジョンの際のプローブを仮定していないので、インデックスを用いた ROT の場合よりもいくらか余計なコストを生じているし、完全にハッシュを利用した場合よりもコストが低くなっているので少々中途半端なモデルである。

この結果から、P3L で採用した方法が十分有効であること、PS-Algol の方法も最適化によってかなり改善されること、ROT を用いる方法のコストがかなり高くなることがわかる。P3L での改良が払う代償としては、プログラムの起動時に全てのポインタが有効な値を保持していることを保証するために Persistent root から直接参照されている永続オブジェクトをユーザが記述した部分の実行前に主記憶内に読み込んでおく必要があることである。しかし、このオブジェクトの数は少なく、P3L のコンパイラが自動的に main の実行開始直後に読み込み用のコードを生成するので特に問題は生じない。むしろ、このような事前の読み込みを行なわないとプログラム中で静的に割り当てられたポインタを二次記憶上のアドレスでリンク時に初期化する必要が生じ、通常のツールを利用して実装を行なうことが難しくなる。

5 仮想空間の回収

最後に仮想空間の回収の問題について考えたい。これまで述べてきたように当初二次記憶上に置かれていたオブジェクト群は参照されるに従って漸進的に主記憶中に読み込まれていく。アプリケーションがアクセスするデータの総量よりも主記憶が大きい間は問題が生じないが、割り当てられた実メモリのサイズを越えた段階からページングが始まる。仮想記憶の機構にたよってアプリケーションの終了までオブジェクトを読み込み続けることも考えられるが、もしも必要のないオブジェクトが残るためページングの影響が大きくでるであろうし、データベースの全件検索を行なうアプリケーションや画像を扱う応用ではバッキングストア、場合によっては仮想空間そのものが不足することが考えられる。

以上のような理由から Persistent Language には仮想空間の回収のための機構が不可欠であると考えられる。またプログラマの生産性向上という観点からは自動的なメモリ管理を行なって、プログラマの負荷を低減することも望ましい。

以下、少々まとまりに欠けるがこの仮想空間の回収についての問題点をいくつか述べる。

5.1 Garbage Collection vs Stash Collection

記憶領域の解放（管理）技術としては Garbage Collection が古くから LISP や Smalltalk の実装において研究されてきた。Persistent Language においても実行時において生じるゴミの

回収に利用できるであろう。しかし、Persistent Languageの特異性は、ゴミがなくとも仮想空間が溢れることである。永続なオブジェクト中にゴミが発生する頻度は、計算の中間過程に生じる一時的なオブジェクト中の場合よりもずっと低く、persistent rootに参照されているオブジェクトから到達不可能になる永続オブジェクトは非常に少ないと考えられる。従ってGarbage Collectionとは全く別の機構で記憶領域を回収する必要がある。それにはPS-Algolでも部分的に採用されたStash Collectionという方法がある。この方法ではアクセスされそうもないオブジェクトを主記憶中から強制的に追い出す。ゴミではないので当然そのオブジェクトへの参照も存在する。そのためなんらかの方法で、後に不正な参照が行なわれないことを保証しなくてはならない。ROTを採用しているのであれば、テーブル中の主記憶アドレスを無効にする方法が考えられる。ポインタから必ずオブジェクトの先頭を取り出せるならば、オブジェクトの先頭にフラグをつけて、追い出した後に墓石を置いておく方法もある。あるいは、＜オブジェクト,そのオブジェクトへの参照のセット＞をサイドテーブルで管理して、追いだし時に参照しているポインタを書き変えるという方法も考えられる。（サイドテーブルの管理にリファレンスカウント以上のCPUコストとかなりのメモリコストがかかること、参照がCPUレジスタ中にないことを保証する機構あるいは、参照が格納されているレジスタを特定する機構が必要になることの注意）参照の書き変えが不可能であったり、サイドテーブルの管理のコストが余りに高い場合には、[Bartlett]のconservative GCの考え方が利用できる。この方法は参照が残っている（可能性のある）オブジェクトには一切触れないで、ゴミであることが（プログラマとの契約上）確定できるオブジェクトのみ回収する方法である。ただ既に述べたように永続オブジェクトへの参照は決して消滅しないと考えた方がよいので、どこかの参照を書き変える必要がある。幸い永続オブジェクトの型は明らかになっているので、永続オブジェクトからしか参照されていないオブジェクトを追いだし、それへの参照を書き換え、一時的なオブジェクト、レジスタあるいはスタックからの参照の残っているオブジェクトには手を触れないことでStash Collectionが可能になりそうである。

5.2 segregated heap or combined heap?

前節の議論から、どうもヒープは一時オブジェクト用のヒープと永続オブジェクト用のヒープに分けたほうが良さそうである。問題点としては、Garbage Collection, Stash Collection時に別のヒープからの参照を考慮する必要があり操作が複雑化する恐れがある。

5.3 problem of heap fragmentation

仮想メモリの管理を考える上で問題になるのがヒープの断片化である。LISPを用いた記号処理のようにオブジェクトのサイズが比較的均一で小さい場合には問題にならないかもしれないが、画像など大きなをデータを含む場合には注意しなくてはいけない。できればcompacting GCができることが望ましい。ただその場合Generationを基にした方式を用いたとしてもコピーのためのコストが0にはならないのでトレードオフを十分見究める必要がある。断片化でもう一つ見逃してはならないのはpersistent object用のヒープのページ内断片化である。通常二次記憶上でオブジェクトはページあるいはセグメント単位でまとめて格納されている。ある瞬間に必要となるオブジェクトはその内の一つであるから、その中から必要なオブジェクトを選択してpersistent object用のヒープにコピーする方法(object at a time)とページ毎コピーする方法(page at a time)の2つが考えられるが、オブジェクトがうまくクラスタリングされていれば当然後者のほうが効率がよい。またうまく実装すれば実際にコピーを行なわないでページテーブルの書き変えで済ますことも可能である。しかし逆にクラスタリングがうまくいっていない場合には、ページのほとんどが無駄に占有されてしまう。アプリケーションの実行以前にクラスタリングの程度を調べることは難しいと思われるので、最初はpage at a timeで読み込みを続け、persistent object用のヒープが不足した段階でページ内断片化の再利用を図る必要があろう。この場合にもcompacting GCが有効である。ただupdate対象のオブジェクトを含むページはそのまま残しておくのが望ましい。ページの中身を一部での捨ててしまうとオブジェクトの更新書き出し時にもう一度おなじページを読み出す必要が生じるからである。

5.4 reclustering of persistent objects

実メモリを意識したstashingを行なわない場合には、仮想空間中でのオブジェクトのリクラスタリングを行なうことでTLBのヒット率向上やpagingのためのディスクのI/Oやシーク時間の低減を図ることができると考えられる。リクラスタリングによるI/O回数の低減はページ内断片化の除去により得られる利点であり、前節での議論と同一のものである。一方リクラスタリングによりオブジェクトをディスク上の近い位置に再配置することによりどれほどの性能向上が得られるのかはよくわからない。実メモリの量を意識し、pagingが始まるまえにStashingを開始する場合には、ページのリクラスタリングには意味がない。リクラスタリングをするとすれば二次記憶上へStash outするときに行なうことになろう。Compacting GCを用いた時のリクラスタリングの効用については[BUTLER]が詳しく述べている。

```
[+-+][-++][+++][-+-][+--][---]
  ==> [+++][+++][+++][---][---][---]
 Removal of Intra-page fragmentation
   (clustering objects in pages)

[+++][---][+++][+++][---][---][---][+++]
==> [+++][+++][+++][+++][---][---][---]
         re-clustering of pages
```

5.5 objects are much bigger than Lisp objects

ゴミの回収にはLISPやSmalltalkでの研究の成果が応用できるであろうと述べた。しかしながらLISPやSmalltalkに比べて大きなオブジェクトを扱う場合には、当然Reference countingとcompacting gcのトレードオフポイントも異なるであろうし、オブジェクトのサイズによってヒープ、そしてGCの方法を変えるといったことも必要になる可能性もある。

5.6 not all objects are typed or tagged

C言語では、LISPやSmalltalkのようにデータ(OID, ポインタ)にタグをつけて参照を整数などのデータと区別したり、Pascalのように強い型付けを行なってヒープ上のオブジェクトの型もきちんと管理するということが行なわれない。ヒープ上にオブジェクトが作られる時には、単にその型のデータを保持するのに十分なメモリ領域を割り当てるだけである。このことはGCを行なうときに十分な考慮を要する。ROTによる参照がおこなわれていれば、存在する参照の全てを把握することができるので問題はない。一方、絶対アドレスを用いている場合には、フォワーダを置いたり、conservative gcを行なうことを考えなくてはいけない。

6 おわりに

以上、Persistent Programming Languageを実装する場合の主記憶空間の管理とくにオブジェクト識別子の表現方法について考察を行なった。今後、メモリ空間の回収についてより深く考えるとともに、garbage collectorやstashing collectorの実装および評価を行なっていく予定である。

[Smith87] Karen E.Smith and Stanley Zdo:
"Intermedia:A Case Study of the Differe:
Between Relational and Object-Oriented
Database systems", OOPSLA87

[3rdGDBM] The comittee for Advanced DBM
Function,"THE THIRD GENERATION DATABASE
MANIFESTO",April 1990,Colledge of
Engineering UCB.

[PS-ALGOL] M. Atkinson et,al."Algorithm
for a Persistent Heap", Software Practi
and Experience Vol 13,1983.

[BARTLETT] Joel F.Bartlett,"Compacting
Garbage Collection with Amibiguous Root
DEC Western Research Laboratory.

[BUTLER] Margaret H.Butler,"Storage
Reclamation in Object Oriented Database
Systems", SIGMOD 87