圧縮型ガーベッジコレクションの高速化について

新田 寛 寺島元章†

圧縮型ガーベッジコレクション（mark-and-compact GC）の処理の高速化に使用した技法とその評価について述べる．古典的な圧縮型 GC はその処理時間が GC 対象領域の容量に比例することから，使用中オブジェクト量が少ない場合や大きな記憶領域を使用する場合は複写型 GC よりも時間的に劣勢にあると見なされてきた．最近の圧縮型 GC は使用中オブジェクトあるいはその塊（クラスタ）のアドレスをソートすることで複写型 GC とほぼ同じ時間量で処理を行うことができる．この種の圧縮型 GC でも，印付けやポインタ補正，ソート処理のために複写型よりも多くの処理時間を要する．本稿では，ソート対象データの個数を減らすことと，ポインタ補正を効率的に行うことでさらなる高速化が得られることを示す．前者はソートの処理だけでなく，印付け処理の負荷も軽減できる．後者は時間的かつ領域的な利得を生む．これらの高速化技法により，比較的大きな容量のヒープでも圧縮型 GC が効果的に使用できるようになった．また，GC 処理時間がプログラムの実行に要する総処理時間の長短に必ずしも直結しないことも示す．圧縮型 GC が複写型 GC と比べて総処理時間では優位になるプログラムが多くある．これは圧縮型 GC の利点である位置に関する局所性の保存と固定容量記憶に対する効率性の良さに起因するものと考えられるが，こうした解析結果についても述べる．

Fast Mark-and-compact Garbage Collection

Hiroshi Nitta and Motoaki Terashima†

In this presentation, we describe garbage collection (GC) schemes used in an implementation of fast mark-and-compact GC and its effects. The traditional mark-and-compact was regarded as inferior to copying GC in its execution time in case of few objects being in use or a large storage space, because the execution time of mark-and-compact GC is proportional to the size of storage space being processed. The mark-and-compact GC that has been developed recently can process its task with the same time complexity as the copying GC by means of an effective technique that sorts address data of objects in use (or clusters made of these objects). Nevertheless such mark-and-compact GC requires more time than the copying GC, due to its phases of marking, sorting and pointer adjustment that are costly in time. We point out that the speed-up of the mark-and-compact GC described above can be performed by decreasing the number of data being sorted and refining a scheme of the pointer adjustment. The former has a good effects on the execution time of both marking and sorting phases, and the latter does on both time and space. They enables the mark-and-compact GC to use a large size heap. The execution time of GC does not necessarily have much effect on the total time of program execution. The mark-and-compact GC the total time of many programs shorter than the copying GC. This fact may result from the merits of the mark-and-compact GC: locality of data objects in the heap and efficiency of the heap utilization.

1. はじめに

ガーベッジコレクション（以下，GC と呼ぶ）は動的データを扱う言語処理系の必須の機能として，これまでに数多くの実装方式の研究が行われてきた[1),2)]．GC は，ヒープに作られたデータオブジェクトでもう参照されることのない使用済みのオブジェクトを回収し，それらが占めていた領域を再利用する自動記憶管理機構のことである．

本稿では圧縮型（mark-and-compact）GC の処理の高速化について述べる．汎用計算機での実装に最適な停止回収型 GC☆は圧縮型 GC と複写型（copying collection）GC[3)] に大別される．圧縮型 GC は複写型 GC に比べて，GC 処理時間が長い点で不利とされてきたが，最近の圧縮型 GC の高速化に関する研究[4)~6)] は，$O(A)$ の時間計算量（time complexity）

† 電気通信大学大学院情報システム学研究科
Graduate School of Information Systems, University of Electro-Communications

☆ GC が起動したら，その一連の処理が終了するまで他の処理（純計算）が待たされるもののこと．実装が比較的容易で，純計算に対する負荷が少ないといわれる．

を持つ高速圧縮型 GC を生み出している．ここで，A は使用中データオブジェクトの総容量である．この種の GC でも，処理の高速化という見地からはまだ改良の余地はいくつかある．それらは次の 3 つである．

(1) ソートのサイズを小さくすること．
高速圧縮型 GC は，大小順になった使用中データオブジェクトのアドレスを使用することで，ヒープに散在する使用中データオブジェクトだけを走査する．ソート処理はこの $O(A)$ 時間計算量を実現する要である．ソート対象となるアドレスの個数を減らすことは時間と領域のコストを減らすことに直結する．

(2) ポインタ補正を高速にすること．
圧縮型 GC は移動対象のデータオブジェクトを指すポインタの補正が時間や領域のコスト増となる．既成の方法[2),7),8)] でも 1 つのポインタを定数時間で補正できるが，複写型 GC の「間接ポインタ」を利用した補正よりも時間を要する．複写型 GC のポインタ補正法を部分的に利用することで，時間のコスト減を図ることができる．

(3) 局所化を行うこと．
データオブジェクトの配置での局所化や作業スペースの共有はワーキングセットサイズを小さくし，キャッシュ記憶の効果的な使用を可能にする．これはキャッシュ記憶を搭載する計算機での処理時間の短縮に直結する．

圧縮型 GC は処理時間の弱点を除けば，記憶領域の効率的な使用ができることや各データオブジェクトがつねにその作られた順序で並び続けるという利点を持つ．後者は生成順序[7)] と呼ばれ，これを利用した実例として，WAM に基づく Prolog 処理系[9),10)] がある．この処理系は任意の 2 つの選択点の生成順序が恒久的に保存されることを利用している．そのどちらが新しいかは単にそれらのアドレスを比較することで求まるからである．

また，作られた順序が変わらないということは，幾多の GC 処理を経て，古いデータオブジェクトは領域の一端に，そして新しいデータオブジェクトはその他端方向に配置されることを意味する．これは多世代の世代別 GC[11),12)] を「自然に」実現することになる[13)]．それらの世代を区切るのは単に 1 つの変数の値である．

本稿では，前述の 3 つ高速化技法とその評価について述べる．高速化した圧縮型 GC は PHL（Portable Hashed Lisp）翻訳系[14)] の実行環境下で実装し，ベンチマークプログラムを実行した．評価はその結果に基づいたものである．さらに，この圧縮型 GC と複写型 GC に基づく 2 世代の世代別 GC の比較結果も述べる．

2. 圧縮型 GC

2.1 生成順序の保存

圧縮型 GC の大きな特徴は生成順序の保存である．筆者らの知る限り，データオブジェクトの再配置にともなう生成順序保存が効率的に行えるのは圧縮型 GC だけである．ただし，生成順序の保存には利得も損失もある．

利得の 1 つは位置に関する局所性である．ヒープ上の任意のデータオブジェクト間の距離は GC を経ても決して遠くにはならないということである．もう 1 つは，順方向にデータオブジェクトはつねに新しくなることである．順方向というのは新たなデータオブジェクトが生成される位置を指すアロケーションポイントがヒープを進む方向である．また，データオブジェクトが古いか新しいかは単にそのアドレスで判定できることも利点である．

損失は GC の負担増である．圧縮型 GC は，GC 対象領域中のデータオブジェクトを使用中のものと使用済みのものとに分類してから，前者を 1 方向に詰めるように再配置する．オブジェクトが再配置されると，それを指すポインタも補正される．圧縮型 GC は，こうした処理の過程で少なくとも 1 回は GC 対象領域を走査する[8),15)]．複写型 GC がデータオブジェクトの参照関係を基にオブジェクトを移動して終わるのとは対照的である．時間計算量（time complexity）でいうと，複写型 GC が $O(A)$ であるのに対して，圧縮型 GC は $O(S)$ となる．ここで，A と S はそれぞれ，GC 対象領域中の使用中オブジェクトの総量と GC 対象領域の容量である．通常，A は S よりかなり小さいので，複写型 GC が時間的に優位になるのである．

2.2 高速圧縮型 GC

GC 対象領域の走査がそのフィールド☆を逐一順番に調べるのではなく，使用済のデータオブジェクトが占めるフィールドを飛ばすことができれば時間量は複写型と同じ $O(A)$ となる．そこで考案されたのが，使用中データオブジェクトの先頭フィールドのアドレスを大小順にソートし，その結果（ソート済みアドレス）を利用する走査法である．以後，これをソート法と呼ぶ．

☆ データオブジェクトが構築されるヒープの最小構成単位のこと．一般に 1 語（32 ビット）が相当する．

表 1 圧縮型 GC の処理時間の比較
Table 1 Comparison of processing time of mark-and-compact GC.

Program	load (α)	sort (n)	Traditional		Sort version	
			T	Tgc	T	Tgc
Lisp	0.6 m	10.3	3.26	0.217	3.12	3.5 m
Tarai-6	0.3 m	10.8	3.29	0.220	3.12	6.1 m
list ver.	0.2 m	11.6	3.28	0.219	3.18	15.m
Reduce	0.053	3498	20.0	0.310	20.1	0.218
Fourier	0.043	5856	20.1	0.284	20.1	0.214
analysis	0.038	7160	20.1	0.305	20.1	0.228
Reduce	0.097	4756	26.3	0.263	26.4	0.335
Differ-	0.091	8729	26.3	0.239	26.6	0.306
ential	0.068	10199	26.2	0.207	26.5	0.192
Reduce	0.170	6175	22.8	2.227	25.6	4.719
Matrix	0.106	8521	22.6	1.665	26.6	5.414
6×6	0.079	11716	22.2	1.427	27.4	6.274

（注） Program 欄の上段，中段，下段はそれぞれ 2 MB，4 MB，6 MB の GC 対象領域で測定．m は千分の一を表す．
T は総処理時間，Tgc は GC 時間，ともに単位は秒．

ソート法による GC の時間計算量は，

$$O(A) + O(n \log n) \quad (1)$$

となる．n はソート対象のアドレスの個数である．

ソート法の開発過程で，n は個々の使用中データオブジェクトのアドレス[5] からクラスタのアドレス[4] に置き換えられた．クラスタとは使用中データオブジェクト群が作る連続したフィールドである．クラスタの個数は使用中データオブジェクトの個数より少ないが，GC 対象領域が大きいとか占有率（load factor，GC 対象領域量に対する使用中データオブジェクト総量の比率）が大きい場合にはかなりの量になる．結果として，ソートに要する時間コストが表面化する．

圧縮型 GC は使用中データオブジェクトを類別するために印を付ける．この処理を印付け（marking）と呼ぶ．印付けはフィールド自体に行う場合と，MBT（Mark Bit Table）と呼ばれる領域に行う場合がある．

表 1 は，GC 対象領域内のすべてのフィールドを走査する古典的（traditional）GC とソート法による（sort version）GC の処理時間の比較である．両者とも印はフィールドに付ける方式である．占有率（α）が非常に小さい Tarai 関数[16] のようなプログラムの実行ではソート法が優位であるが，数式処理システム Reduce[17] のプログラムでは優劣が逆転するものがある．それらは総じてクラスタ数（n）が多い．使用した処理系 PHL の概要や計算機の仕様については後で述べる．なお，これらの GC は現在使用に供されてはいない．

2.3 世代別管理

圧縮型 GC は比較的容易に世代別 GC になる．世代別 GC とはデータオブジェクトの寿命に関する仮説に基づいてオブジェクトをその寿命によっていくつかの世代に分け，minor collection よりも低い頻度で major collection を起動させて，GC 処理時間の短縮を意図した手法である．

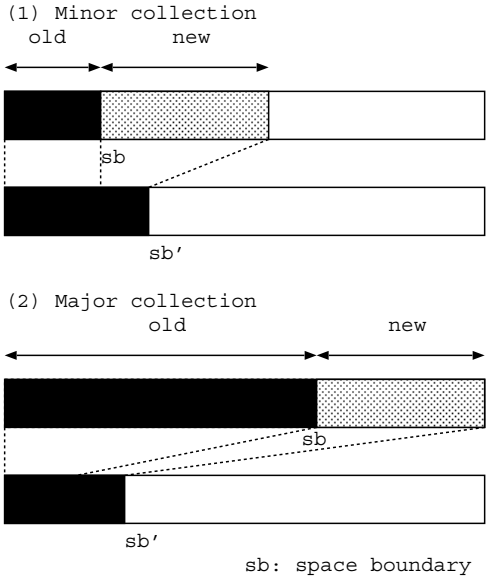

図 1 圧縮型二世代 GC
Fig. 1 Two generational GC of mark-and-compact.

図 1 は圧縮型 GC が行う新旧二世代のデータオブジェクト管理の様子である．GC は機能的に minor collection と major collection に分かれるが，両者の違いは単に GC 対象領域が異なる（後者の方が広い）だけである．minor collection は，ヒープの一部である新世代領域（図中の new）が消費されたとき起動される．使用中オブジェクトはヒープの左側に詰められ，古世代領域（図中の old）を形成する．世代別 GC でいうところの「1 回の GC 経験で殿堂入り」するのである．新世代と古世代の境界は可変で，1 つの変数（図中の sb）がその位置を決める．次の新世代領域は古世代領域に隣接してとられるが，その容量を可変にすることもできる．

また，GC 経験回数を複数にして殿堂入りさせることもできる．2 回にした実験報告もあるが，特に時間的な優位性があるわけではない[13]．

major collection が起動されるのはヒープが新旧二世代の領域で一杯になるときである．その GC 対象領域はヒープ全体である．minor collection と同様に，使用中オブジェクトはヒープの左端に詰められて，古世代領域を形成する．

既成の圧縮型世代別 GC[13),15),18)] では，新世代領域を対象計算機の二次キャッシュサイズに合うように

設定している．これはヒープよりもかなり少ない容量である．その理由は，頻繁に使用される新世代の領域がキャッシュ記憶に置かれれば，キャッシュヒットの効果により処理の高速化が期待できるからである．

3. 高速化手法

ここでは，前章で述べたソート法による圧縮型 GC の処理時間をさらに短縮する手法について述べる．前提として，GC は次の機能を有するものとする．

(1) 二世代の世代別 GC
理由は圧縮型 GC が実現する最も単純な世代別 GC であり，利得も十分期待できるからである．

(2) MBT の使用
印付けを MBT で行う理由は，印付け情報が 1/32 に凝縮されるため，クラスタの両端や補正値の算出でのメモリ参照が局所的になり，処理時間の短縮が図られるからである．

(3) スタックの陽な使用
リストの印付けやソートの再帰は陽に記述したスタックを使用する．理由は時間と領域コストの削減のためである．

なお，新世代領域の容量は固定であり，データオブジェクトは GC1 回の経験回数で殿堂入りし，ヒープ全体が消費尽くされたときに，major collection が起動する．アロケーションポイントはヒープをアドレスが大きくなる方向に進むものとする．

3.1 ソ ー ト

MBT の 1 語は 32 個の連続したフィールドの印の情報を持つ．これをエントリと呼び，そのフィールド群を小区画と呼ぶ．小区画内ではアドレスの小さいフィールドがエントリの下位のビットに対応している．

1 つの小区画に対して登録するクラスタをたかだか 1 にすれば，ソートされるデータの個数を減らすことができる．走査方向を考えると，小区画内のアドレスの最も小さいもの（エントリでは下位のビット）が最適である．同じ小区画内の他のクラスタはエントリに対するビット演算で順に求めることができる．同じ理由で，直前の小区画から続くクラスタがある場合，その小区画内のクラスタは登録しない．こうして，細かなクラスタが多数散在する場合のソートコストをかなり軽減できる．

次の疑似的プログラミング言語による記述と**図 2** はこの登録処理を説明したものである．データオブジェクトに印を付ける手続き mark は説明の簡潔さのために CONS 型データ以外の処理と世代別管理の処理が省かれている．

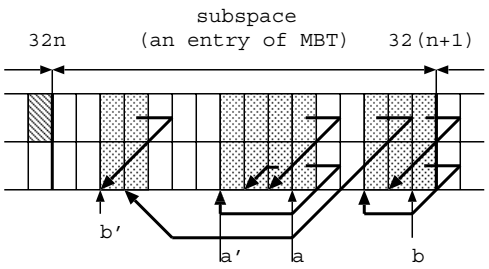

図 2 ソート対象アドレスの登録
Fig. 2 Registration of address data being sorted.

```
procedure mark(p)
if p is a terminator or already marked then return;
else if p is a cons object then
  begin
  mark two bits of MBT corresponding to p;
  increment number_of_marked_bits by 2;
  mark(car(p)); mark(cdr(p));
  if ¬marked_beforep(p) then store D with p;
  end;

function marked_beforep(p)
if any bit lower than p in a p's entry is marked or
   the MSB of the entry just before p's is marked
   then return true;
else return false;
```

CONS 型データの処理では，まずその CAR と CDR のフィールドに対応する MBT の 2 ビットに印が付く．印の累計値が 2 つ増分されるが，これは次節で述べる手法で使用される．次に CAR と CDR の処理が行われる．この順序は新規に CONS 型データが作成される場 合は CAR のデータが CDR のデータよりも前に作られるという事実☆を考慮している．最後に，印情報が marked_beforep で検査される．この検査は，p のエントリ内で p よりも下位のビットに印があるか，あるいはその直前のエントリの MSB に印があるかを調べる．この結果が「否」ならば，p をソート対象データ（D）として登録する．

図 2 は 1 つの小区画を含むヒープの断片を模式化したものである．上段と下段は CONS 型データの CDR と CAR の各フィールドである．ポインタ表示がないフィールドは終端子であり，右にアドレスが大きくなる．このとき，mark(b) の実行で S に登録されるアドレ

☆ CAR と CDR がヒープを指すポインタであるとき，CAR の方が小さいアドレス（遠くを指す）となる．入力部で使用されるパーザや評価系の evlis の生成データはこの事実と合致する．

スは b' の 1 つだけである．それ以前に mark(a) が実行されていれば，a' が登録されているので 2 つになる．ただし，登録された全データに対して，marked_beforep による再検査が行われるので，a' のようなデータはソート処理から除かれる．もし，直前のエントリの MSB（図中の斜線で示した CDR フィールドに対応するビット）に印があったならば，このエントリにある a' や b' のようなアドレスは何も登録されない．

以上は，1 つの小区画に対してソート対象のクラスタのアドレスをたかだか 1 個にすることであったが，前述の手続き marked_beforep を変更すれば，アドレスを絞る区画の単位を 64 語やそれ以上，あるいは 16 語や 8 語にすることもできる．

まとめると，印付けの段階は次のようになる．

```
begin
clear MBT;        // set 0 to its bits.
for p in root_set do mark(p);
for p in D do
  if marked_beforep(p) then delete p from D;
sort D;
end;
```

MBT は補正表と両者のエントリが交互に並ぶように対で記憶領域にとられる．その最大容量は両者を合わせてヒープ容量の 1/16 である．印付け段階では使用されない補正表の広いスペースは mark の再帰とソート対象データの登録に使用される．

3.2 ポインタ補正

ポインタ補正は，補正表と MBT を用いた表の計算から求めるか，フィールドに埋め込まれた再配置先ポインタをそのまま利用するかのいずれかで行われる．前者は従来の方式[7),13)] であるが，後者は複写型 GC で forward pointer と呼ばれている方式である．ともに，1 つのポインタ補正を定数時間で行うことができるが，後者の方が時間を要しない．この節では，再配置ポインタを効果的に使用する方策について述べる．

圧縮型 GC では，使用中データオブジェクトの再配置にともない，一部のオブジェクトが上書きされる．たとえば，**図 3** (2) の sb と sb' 間に存在するデータオブジェクトがそれである．これらのオブジェクト（使用中のもの）を単オブジェクトと呼ぶ．再配置後のオブジェクトだけが存在するからである．単オブジェクトに対しては補正表を作り，MBT との表計算から補正値を求める．それ以外の使用中オブジェクトを複オブジェクトと呼ぶ．元の（再配置前の）オブジェクトも残るからである．そこで，図 3 (2) の A が示すように，元のオブジェクトを再配置先を指すポインタで書き換えると，複オブジェクトのポインタ補正は間接参照で行えることになる．単オブジェクトの比率が小さければポインタ補正全体に要する時間も短縮されるはずである．

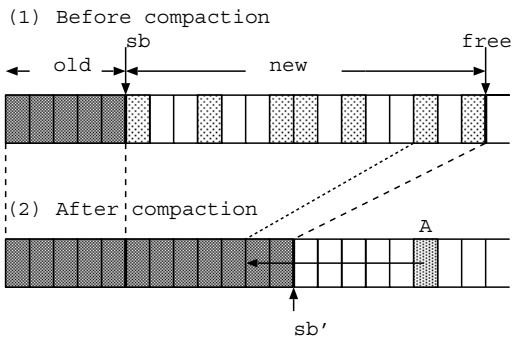

図 3 オブジェクトの再配置
Fig. 3 Relocation of objects.

3.2.1 再 配 置

使用中データオブジェクトの再配置では次の 2 項が同時に行われる．

(1) 補正表の各エントリの作成，あるいは再配置先ポインタの置き換え
(2) 逆向きポインタの補正

補正表の原理は各小区画の先頭フィールドの補正値（そこのオブジェクトが再配置で移動する距離）を 1 つのエントリとして持ち，この補正値と小区画内の印付け情報で個々のデータオブジェクトの補正値を定数時間で求めることである．補正表も二分探索の導入で[15)]，該当する MBT の 8 ビット分で補正値が算出できるようになってはいるが，時間的には再配置ポインタ（間接ポインタ）に匹敵するものではない．補正表が作られるのは，図 3 の sb' がある小区画までである．この地点をあらかじめ算出するのに印付けオブジェクトの総容量が必要なのである．

sb' 以降のオブジェクトには再配置先を指すポインタが埋め込まれる．ただし，CDR フィールドやシンボルの印字名など単独で参照されないものはすべて除かれる．これを実現するために，フィールドからそのオブジェクトの型が判定できるような微細なデータ表現☆の変更が行われている．

逆向きポインタとは新しいオブジェクトから古いオブジェクトを指すポインタのことである．この性質から，起点のオブジェクトの再配置時に，そのポインタ

☆ いわゆるポインタタグでは，フィールドの情報だけでそのデータオブジェクトの型は分からない．

が指すオブジェクトの補正表か再配置先ポインタができているので，オブジェクトの再配置と逆向きポインタの補正が同時に行えるのである．一般に「新しいものは古いものから作られる」ので，前述の前向きポインタよりもかなり大きな個数になるはずである．一方，前向きポインタはこの段階では補正できないので，それらを補正表の未使用の部分に登録しておく．再配置が完了して，補正表や再配置先ポインタがすべて作られてから，前向きポインタの補正が行われる．前向きポインタを登録するのは，再配置されたオブジェクトの領域を再度走査しないための策である．

印付けに MBT などの「外部表」を使用する場合，GC 対象領域を走査する回数は 2 回で済む．今回はこれを 1 回にしたことになる．ただし，前向きポインタを記録するための領域がなくなると，GC 対象領域の一部を走査することになるが，こうした事態は生じていない．

3.2.2 後 処 理

ポインタ補正は，スタックや OBLIS，リメンバードセットなどのルートセットの処理で終える．ポインタ補正が終了した時点で，リメンバードセットは空になる．

4. 実験と評価

4.1 実 験 環 境

本 GC を含む各 GC の性能評価は，それらを組み込んだ PHL 翻訳系の実行環境下で行い，言語やその処理系の特性に依存しない普遍的な結果を得ることを目標とした．こうした環境では，純粋に機械語のプログラムが生成するデータオブジェクトのみが GC の対象になるからである．なお，PHL は可搬性のある Lisp 処理系で，これまでに，SPARC，Alpha，MIPS，Pentium，MC68000 などの命令セットアーキテクチャを持つ計算機で稼働している．

今回使用した計算機は Sun Fire 280R（Sun Microsystems 社製）で，900 MHz の Ultra SPARC-Ⅲ Cu を 2 個搭載する．二次キャッシュ容量は 8 MB で，主記憶容量は 4 GB である．オペレーティングシステム（OS）は日本語 Solaris8 である．C コンパイラは gcc-2.95.3 である．処理時間の測定は gethrtime()[19] による．ナノ秒単位の時間計測ができるが，MPU 時間そのものを返すので，他のプロセスの影響を極力排除するようにした．もう 1 つは Dell Precision 530（デルコンピュータ社製）で，1.5 GHz の Intel Xeon を 2 個搭載する．二次キャッシュ容量は 256 KB で，主記憶は 1 GB である．OS は Red Hat Linux 9 で，C コンパイラは gcc-3.2.2 である．処理時間の測定はこれも MPU のクロック数を返す rdtsc 命令を利用した．

使用したプログラムは，Lisp で書かれた Tarai 関数の他はすべて Reduce[17] プログラムである．Tarai-6 list ver. はリストを引数とする Tarai 関数[16] である．整数の大小をリストの長短に対応させることで長大な再帰計算を行うものである．

Reduce は A.C. Hearn 教授らが開発した数式処理システムで，不定積分や因数分解のパッケージを除いた基本部分は Lisp ソースにして約 4,000 行，翻訳すると約 600 KB（UltraSPARC の機械語）になる．Reduce プログラムは，Reduce が読み込み，解釈し，結果を出力するという形式で実行される．この実行過程で生成されるオブジェクトの特性は数式や計算内容で大きく変わる．個々のプログラムでは，Differential は多項式の数式微分，F and G series 33 は F&G 系列の 33 項までの算出である，Fourier analysis はフーリエ解析計算，Matrix 6×6 は Matrix は 6 行 6 列の逆行列の算出とその検算，Bignum vector は多倍長整数計算である．

4.1.1 ソートのコスト

表 2 は 3.1 節で述べたソート対象データ個数（n）を減らすことが GC の実行時間（Tgc），とりわけソート処理の時間（Ts）の短縮にいかに寄与しているかを示している．各プログラム欄の上段は旧版で，中段はソート対象データ数を減らした改良版で，32 語の小区画でアドレスを絞ったもの，下段は同じく 2 つの小区画をまとめて 64 語の単位でアドレスを絞ったものである．n の左側は記録されたデータの個数の平均値で，右側の実際にソートされたデータ数の平均値に近いほど，無駄なくデータが記録されたことになる．

プログラムにも依存するが，旧版（上段）から 32 語単位版（中段）ではデータ数が急激に，最大で 1/3 に減少し，GC 処理時間の短縮が実現されている．32 語版から 64 語版（下段）では緩やかに，最大で 2/3 に減少する．データ数が依然として多いプログラムは GC 処理時間が短かくなるが，時間がほぼ同じか，逆に増加するものもある．表 2 には記載されていないが，新世代領域が 4 MB であるとき，96 語の単位でアドレスを絞ると，Tarai は 7.4，Fourier は 1650，F&G は 1081，Diff. は 2153，Matrix は 1143，Bignum は 440 がソート対象データ個数となる．

結論として，32 語の小区画でソート対象データを 1 つに絞り込むのは効果的であったといえる．

表 2 の上半分は新世代領域を 2 MB に，下半分は 4 MB にしたときの測定値である．ヒープの容量は

表 2 ソート法の GC 処理時間の比較
Table 2 Comparison of GC processing time on sorting.

Program	α	n		Tgc	Tm	Ts
new genaration space : 2 MB						
Tarai-6		10.3	10.3	8.6 m	7.6 m	0.07 m
list ver.	0.6 m	7.3	7.2	8.3 m	7.5 m	0.06 m
		7.2	7.1	8.0 m	7.0 m	0.07 m
Fourier		3749	3498	0.181	0.080	0.047
analusis	0.053	1831	1536	0.146	0.078	0.010
		1448	1130	0.142	0.075	0.008
F and G		2277	1472	0.120	0.079	0.008
series	0.067	1488	846	0.115	0.075	0.004
		1148	651	0.117	0.077	0.003
Differ-		5026	4756	0.303	0.131	0.093
ential	0.097	2552	2130	0.172	0.086	0.026
		1859	1438	0.153	0.081	0.010
Matrix		9430	6175	3.852	0.873	2.480
6×6	0.170	4053	1839	1.656	0.815	0.325
		2700	1134	1.405	0.770	0.111
Bignum		1032	1025	0.605	0.452	0.056
vector	0.253	1019	975	0.549	0.385	0.065
		1014	654	0.568	0.442	0.026
new genaration space : 4 MB						
Tarai-6		10.9	10.8	0.012	11 m	0.04 m
list ver.	0.3 m	7.9	7.8	9.2 m	8.7 m	0.04 m
		7.6	7.5	0.012	12 m	0.04 m
Fourier		6262	5856	0.133	0.066	0.023
analysis	0.043	3136	2676	0.121	0.066	0.009
		2526	2005	0.118	0.064	0.007
F and G		4520	2917	0.115	0.074	0.008
series	0.062	2937	1675	0.105	0.068	0.004
		2269	1295	0.103	0.067	0.003
Differ-		9304	8729	0.268	0.075	0.143
ential	0.091	4817	4043	0.158	0.077	0.025
		3487	2760	0.139	0.073	0.010
Matrix		11786	8521	4.678	0.519	3.824
6×6	0.106	5016	2593	1.309	0.530	0.406
		3320	1672	1.006	0.507	0.127
Bignum		1056	1032	0.300	0.227	0.025
vector	0.128	1037	950	0.282	0.200	0.033
		1028	663	0.291	0.226	0.013

（注）Program 欄の上段は旧版，中段は改良版（32 語単位），下段は同（64 語単位），α は占有率，n はソート対象データの平均個数（右は候補数），Tgc は GC 時間，Tm と Ts は印付けとソートの各処理時間，m は千分の一を表す，処理時間の単位は秒．計算機は Sun Fire 280R.

8 MB である．両者で占有率（α）とソート対象データ数は異なるが，これらの値が示すように，今回使用したプログラムはその特性に偏りのない多種多様なものである．

4.1.2 ポインタ補正のコスト

表 3 は 3.2 節で述べたポインタ補正の改良により GC の実行時間（Tgc）やそれを含む総処理時間（T）がどのようになっているかを示している．上段がソート法，下段が MBT 全走査法に基づいた改良版の結果である．この改良自体は，ソート法や MBT 全走査法

表 3 処理時間の比較
Table 3 Comparison of processing time.

Program	T	Tgc	Tms	pointers	
new genaration space : 2 MB					
Tarai-6	2.78	0.010	0.009	0	0.18
list ver.	2.80	0.018	0.008		
Fourier	18.4	0.119	0.085	1.35	12.6
analysis	18.0	0.105	0.056		
F and G	8.92	0.099	0.075	1.24	14.3
series	8.71	0.088	0.057		
Differ-	23.9	0.153	0.122	5.83	20.3
ential	23.3	0.112	0.074		
Matrix	22.0	1.40	1.143	15.0	30.1
6×6	21.1	1.00	0.619		
Bignum	15.2	0.457	0.385	0.06	1.08
vector	15.1	0.432	0.342		
new genaration space : 4 MB					
Tarai-6	2.75	0.010	0.010	0	0.19
list ver.	2.83	0.017	0.008		
Fourier	18.4	0.095	0.068	1.35	21.6
analysis	18.0	0.089	0.049		
F and G	8.87	0.088	0.068	2.10	26.6
series	8.78	0.084	0.056		
Differ-	24.0	0.131	0.107	10.3	39.7
ential	23.3	0.101	0.065		
Matrix	21.5	1.060	0.832	7.59	52.8
6×6	20.8	0.661	0.414		
Bignum	15.0	0.247	0.212	0.13	1.08
vector	14.7	0.227	0.174		

（注）Program 欄の上段はソート法，下段は MBT 全走査法 T，Tgc，Tms はそれぞれ総処理時間，GC 時間，印付けとソートの合計時間（単位は秒）．pointers はポインタ補正の平均操作回数（単位は K），他は表 2 と同じ．

というクラスタ処理の手法とは別個である．MBT 全走査法は MBT の情報でクラスタを走査する．アドレスのソート処理は不要であるが，MBT のすべてのエントリが順に調べられる．もちろん，すべてのビットが 0 か 1 であるようなエントリは読んで飛ばす（個々のビットは調べない）ことができるので，MBT を走査する手間がソートにかかわる手間以下ならば時間の短縮が図られる．表 3 の Tms の値から分かるようにソート処理のない分，一般に MBT 全走査法の方が GC 実行時間が短い．総処理時間では Matrix 6×6 を除けば大きな違いはない．

表 3 の pointers はポインタ補正が補正表かあるいは再配置先ポインタのどちらで行われたかを示す．その左側が前者で，右側が後者である．これらの値も GC ごとの平均値で，単位は K である．両者の比率は単オブジェクトと副オブジェクトの容量比と関連するので，占有率が小さいとか新世代領域が大きい場合に，再配置先ポインタを使用した，より速いポインタ補正が多く行われることになる．ソート法の場合，それが GC の処理時間の短縮に直結していることが表 2

表 4 複写型二世代 GC の処理時間
Table 4 Processing time of two generational copying GC.

Size	1 MB		2 MB		4 MB	
Program	T	Tgc	T	Tgc	T	Tgc
Tarai-6	3.25	4.9 m	*3.27*	**2.7 m**	*3.28*	**1.5 m**
list ver.	3.22	6.2 m	*3.26*	**3.2 m**	*3.32*	**1.7 m**
Fourier	18.7	0.227	*18.7*	*0.150*	*18.8*	*0.120*
analysis	19.0	0.132	*18.9*	*0.101*	*18.8*	*0.079*
F and G	9.93	0.163	*9.92*	*0.144*	*10.0*	*0.132*
series	9.96	0.117	*9.96*	*0.096*	*9.98*	*0.082*
Differ-	24.5	0.171	*24.5*	0.135	*24.4*	0.117
ential	24.8	0.113	*24.7*	0.102	*24.7*	0.100
Matrix	23.6	2.19	*23.5*	1.56	*22.1*	0.701
6×6	23.5	1.42	*22.7*	1.08	*22.3*	0.663
Bignum	18.3	1.01	*17.5*	**0.379**	*17.3*	**0.164**
vector	18.0	0.800	*17.6*	**0.421**	*17.4*	**0.220**

（注）Program 欄の上段と下段はそれぞれ殿堂入りの GC 経験回数が 2 回と 1 回，Size は新世代領域（半領域）量，T は総処理時間，Tgc は GC 時間（単位はともに秒），m は千分の一を表す，計算機は Sun Fire 280R.

の Tgc との比較で分かる．

GC 対象領域中の順方向ポインタの個数の平均値は，最小が Tarai 関数の 2 で，最大が Differential の 1326 である．残りは 100 程度と比較的少ない．このため，一般的なプログラムで領域の不足はまず起こりえない．

4.2 複写型二世代 GC

最後に，複写型二世代 GC との処理時間の比較について述べる．本稿で述べた高速化の手法は圧縮型 GC の実行時間や総処理時間をかなり短縮させている．その到達レベルを複写型 GC との比較で論じるのは実用性の見地からも十分に意義がある．比較対象は圧縮型 GC の世代管理に近い複写型二世代 GC である．この GC は「新古各 2 面」方式を採用し，GC を 1 回か 2 回経験した新世代オブジェクトを殿堂入りさせる．なお，新世代領域（半領域）の量を圧縮型 GC の新世代領域と同じにして，1 回の GC で殿堂入させるようにすると，オブジェクトの移動量に関する限り 2 つの GC は同じになる．

複写型 GC の古世代領域（半領域）は 8 MB と比較的大きいが，Matrix と Bignum のプログラムでは major collection（古世代領域間でオブジェクト移動が起こる）が起動された．この点は圧縮型でも同じである．

表 4 の新世代領域（Size）の 2 MB と 4 MB に記載された GC 実行時間（右側）や総処理時間（左側）で，太字は複写型 GC が優位にあるもの，斜字は圧縮型 GC が優位にあるものである．各欄の上下は殿堂入りの条件が異なり，上段が GC2 回，下段が 1 回である．

GC 実行時間では，Tarai 関数のように占有率がきわめて小さいか，Bignum のようにベクタ構造が多用されるときは複写型 GC が優位である．総処理時間では圧縮型 GC が概して優位にある．

表 5 圧縮型と複写型 GC の処理時間の比較
Table 5 Comparison of processing time of mark-and-compact and copying GC.

Program	Tarai-6		Matrix		Bignum	
GC type	T	Tgc	T	Tgc	T	Tgc
sort 2	1.64	14 m	11.4	0.970	**8.55**	284 m
mbps 2	1.65	16 m	**10.4**	**0.689**	8.93	246 m
copy2 2	**1.56**	**2 m**	11.7	1.348	8.76	**194 m**
sort 4	**1.58**	13 m	10.8	0.749	8.74	108 m
mbps 4	1.61	20 m	**10.7**	**0.469**	8.83	132 m
copy2 4	1.59	**1 m**	11.0	0.675	**8.63**	**79 m**
program	Fourier		F and G		Differential	
sort 2	24.2	90 m	7.15	70 m	33.4	112 m
mbps 2	26.0	84 m	6.96	62 m	**33.3**	91 m
copy2 2	23.5	113 m	**5.40**	109 m	34.0	102 m
copy1 2	**23.4**	**70 m**	6.93	**62 m**	37.3	**70 m**
sort 4	25.0	83 m	6.96	66 m	37.1	95 m
mbps 4	26.6	74 m	6.86	60 m	**36.9**	83 m
copy2 4	24.3	93 m	**5.58**	91 m	39.4	87 m
copy1 4	**24.2**	**54 m**	6.83	**53 m**	39.4	**67 m**

（注）sort：ソート法，mbts：MBT 全走査法，copy1：複写型 GC（GC1 回で殿堂入り），copy2：複写型 GC（同 2 回），次の数値は新世代領域量（単位は MB），計算機は Dell Precision 530，他の記号は表 4 と同じ．

表 5 は計算機を Dell Precision 530 に替えたときの処理時間の比較である．アーキテクチャの差異は別にして，量的には MPU の能力は 5/3 倍で，二次キャッシュ容量は 1/32 になる．GC 実行時間に関して両者の優劣に大きな変化はない．しかし，総処理時間では，MPU の処理速度向上効果の得られない Fourier や F and G のプログラムで複写型 GC が優位になった．

5. 関連研究

これまでに，ソート法に基づく GC の開発研究は多方面で行われてきた．しかし，その多くはオブジェクトの再配置のない mark-and-sweep 法に基づいている．最近の事例である Chung らの GC[5] は JVM 上の具現である．ヒープ中に散在する使用中データオブジェクトだけを走査するのにソート法を利用するのであるが，その記述にはクラスタの概念は登場しない．

本稿で述べた高速化手法は，MOA[4] から便宜的 GC[18] を経て，上野らの GC[13] に至る高速圧縮型 GC にすべて適用可能である．ただ，効率の点で，比較的大きな領域を対象にする GC に適しているといえる．

複写型 GC でもオブジェクトを移動する手法として，古典的な幅優先[20] だけでなく，深さ優先[21] や参照関係を優先する方式や，参照をページ内にとどめる

ような局所性を考慮した方式[2]が提案されている．

6. おわりに

本稿では，既成の高速圧縮型GCをさらに高速化するための手法とその効果について述べた．意図したとおり，改良後の圧縮型GCが実行時間や総処理時間で複写型GCより優位になるプログラムが多くなった．これは，GC時間の短縮と，圧縮型GCが本来有するデータオブジェクトの位置に関する局所性の保存と記憶領域の効率的な使用との相乗効果によるものと考えられる．その一方で，Intel Xeonの計算機では複写に有利に働くプログラムが存在する．その解明は今後の課題である．

生成順序を保存するGCが動作する処理系では，プログラマはデータオブジェクトの生成順序を利用した効率的なプログラムが書ける．順序関係の利用はアルゴリズム構築の基本であるので，この順序関係を利用した効率的なプログラムの時間的な利得はGC処理時間の損失を相殺することになるであろう．

参考文献

1) Wilson, P.R.: Uniprocessor Garbage Collection Techniques, Technical Report, University of Texas, Tex. (1994).
2) Jones. R. and Lins, R.: *Garbage Collection*, John Wiley & Sons, UK. (1996).
3) Fenichel, R.R. and Yochelson, J.C.: A Lisp Garbage Collector for Virtual Memory Computer Systems, *CACM*, Vol.12, No.11, pp.611–612 (1969).
4) Suzuki, M., Koide, H. and Terashima, M.: MOA — A Fast Sliding Compaction Scheme for a Large Storage Space, *IWMM95*, LNCS 986, pp.197–210, Springer-Verlag (1995).
5) Chung, Y.C., et. al.: Reducing Sweep Time for a Nearly Empty Heap, *POPL '00*, pp.378–389, ACM (2000).
6) Terashima, M., Ishida, K. and Nitta, H.: The Design and Analysis of the Fast Sliding Compaction Garbage Collection, *Advanced LISP Technology*, Taylor & Francis, N.Y. (2002).
7) Terashima, M. and Goto, E.: Genetic Order and Compactifying Garbage Collectors, *Information Processing Letters*, Vol.7, No.1, pp.27–32 (1978).
8) Morris, F.L.: Time- and Space-efficient Garbage Collection algorithm, *CACM*, Vol.21, No.8, pp.662–665 (1978).
9) Appleby, K., et al.: Garbage collection for Prolog based on WAM, *CACM*, Vol.31, No.6, pp.719–741 (1988).
10) Bekkers, Y., Ridoux, O. and Ungaro, L.: Dynamic Memory Management for Sequential Logic Programming Languages, *IWMM92*, LNCS 637, pp.82–102, Springer-Verlag (1992).
11) Liebeman, H. and Hewitt, C.: A Real-time Garbage Collector based on the Lifetimes of Objects, *CACM*, Vol.26, No.6, pp.419–429 (1983).
12) Ungar, D.M.: Generarion Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm, *ACM Conference on Practical Programming Environments*, pp.157–167 (1984).
13) 上野真由子，山室弥久，寺島元章：圧縮方式による世代別ガーベッジコレクションの実装について，情報処理学会論文誌：プログラミング，Vol.43, No.SIG1 (PRO 13), pp.1–9 (2002).
14) 寺島元章，山本洋司，古川敦司，渡辺美苗：PHLの新コンパイラ，記号処理研究会資料 SYM 78, pp.17–24 (1995).
15) 佐藤圭史，寺島元章：圧縮型ガーベッジコレクションの高速化，情報処理学会論文誌，Vol.40, No.5, pp.2397–2403 (1999).
16) Okuno, G: The Report of the 3rd Lisp Contest and the first Prolog Contest, 情報処理学会研究報告 85–SYM–33 (1985).
17) Hearn, A.C.: *REDUCE User's Manual, version 3.4*, The Rand Corporation, CA. (1988).
18) 宮本崇生，寺島元章：ハイブリッドガーベッジコレクションの実装と評価，情報処理学会論文誌：プログラミング，Vol.40, No.SIG7(PRO 4), pp.24–31 (1999).
19) Mauro, J. and McDougall, R.: *Solaris Internals*, Prentice Hall (2001). 福本　秀ほか（訳）：SOLARIS インターナル．
20) Cheney, C.J.: A Non-recursive List Compacting Algorithm, *CACM*, Vol.13, No.11, pp.677–678 (1970).
21) 八杉昌宏，伊藤智一，小宮常康，湯浅太一：小量のスタックで大部分を深さ優先にコピーするゴミ集め方式，第3回プログラミングおよび応用のシステムに関するワークショップ，SPA2000，日本ソフトウェア科学会 (2000).

(平成15年9月22日受付)
(平成16年1月 8 日採録)

新田 寛

昭和 46 年生．平成 15 年電気通信大学大学院情報システム学研究科博士後期課程を単位取得済退学．プログラミング言語処理系における記憶管理方式に興味を持つ．

寺島 元章（正会員）

昭和 23 年生．昭和 48 年東京大学理学部卒業．昭和 50 年同大学院修士課程，昭和 53 年同博士課程修了．理学博士．昭和 53 年電気通信大学計算機科学科助手．現在，同大学院情報システム学研究科助教授．プログラミング言語とその処理系，記憶管理方式等に興味を持つ．ACM 会員．