ハイブリッドガーベッジコレクションの実装と評価

宮本崇生† 寺島元章†

当研究室で開発したハイブリッド型の高速圧縮型ガーベッジコレクションのLisp翻訳系（PHLC）での実装とその評価について述べる．本ガーベッジコレクション（GC）は，時間短縮効果が期待できる記憶領域の一部を対象に世代別管理に準拠した処理を行う機能と，未処理で残された記憶領域を効果的に回収する機能を合わせ持つものである．それは，使用中データの局所化やそれにともなうワーキングセットの縮小効果などの相乗効果で処理系自体の高速化に大いに寄与する．PHLCはLispプログラムをC言語のソースプログラムに変換した後で，それを対象計算機の機械語に変換する方式の可搬的な翻訳系である．このような実行環境ではGCは機械語プログラムが生成する純粋なデータだけの処理を行うことになる．このため，特定の言語や処理系に依存しないGCの性能評価が可能となる．こうした評価に基づいた本GCの利点についても述べる．

Implementation and Evaluation of Hybrid Garbage Collection

TAKAO MIYAMOTO† and MOTOAKI TERASHIMA†

Hybrid garbage collection, a new type of garbage collection (GC) based on a fast sliding compaction scheme developed in our laboratory is presented with its implementation and evaluation. The hybrid GC has been successfully implemented on a compiler based Lisp system called PHLC, and it has two distinctive features. One is to focus its scavenge on a part of heap to achieve a time saving similar to generation GC, and another is to effectively reclaim remains of the heap being left unscavenged. The hybrid GC improves performance of the Lisp system by localization of data objects being in use and a resultant decrease in working set size. PHLC translates Lisp source programs into C source programs and then they are translated to target machine codes. Consequently the GC processes the data objects that are generated by the execution of machine codes. Therefore performance evaluation of the GC can be done independent of implementations. The analysis of the hybrid GC on its performance is also described.

1. はじめに

ガーベッジコレクション（以下，略してGC）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]．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，その領域を再利用できるようにする自動記憶管理機構である．

今日のように計算機におけるメモリが大容量化し，アプリケーションによるヒープの使用が大規模になると，処理の中断を最小限に押さえるため，処理が速くかつ停止時間が短いより効率的なGCが必要とされる．また，GCの意義もヒープの絶対的不足を解消するという消極的側面よりも処理系を効率的に動作させるという積極的側面が強調されるようになっている．

当研究室で開発された一連の圧縮型ガーベッジコレクション（Sliding Compaction GC）[2)~4)]は従来の圧縮型GCと比べてその処理が速いことで知られている．これらを高速圧縮型GCと呼ぶことにする．その内でも，時に応じて最新の使用領域（ヒープの一部）のみを回収する便宜的GC☆は処理速度が格段に向上している[4]．それは，ヒープに散在する最新の使用中データオブジェクトを効果的に凝縮（圧縮）し，それらの局所性を高めることでスワップを防ぎキャッシングの効果を最大限利用できるからである．さらに，状況次第では，GCを起動せずにプログラムを実行するよりも本GCをタイミング良く動作させてプログラムを実行する方が全体として速くなる．

本稿では，未回収で残された領域を定期的に回収する機能を追加したハイブリッドなGCについて述べ，

† 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

☆ 造語 occasional GC の日本語訳

その利点として，

(1) ヒープのより効率的な利用が図れる，

(2) GC処理を時間的に細分化できる

ことを示す．そして，長期間の実行が求められるような大規模プログラムにおいても処理が効率的に行えることも示す．

また，このGCの性能評価をPHL（Portable Hashed Lisp）翻訳系[5),6)]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．この環境下では，プログラム自身や解釈系（interpreter）が生成する連想リストなどのオブジェクトはヒープに存在せず，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．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CPUを持つ計算機で稼働している．

2. GC

2.1 圧縮型GCの高速化

GC処理の間はそれ以外の処理，いわゆる純計算を停止させる停止回収型GCは単一プロセッサで効率的かつ容易に具現できることから今でも実装法の研究が行われている．停止回収型GCは，複写型GC（Copying GC）[8)]と圧縮型GCの二つに大別される．最近の圧縮型GCの高速化に関する研究は，$O(A)$の時間計算量（time complexity）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は，使用中データオブジェクトのクラスタ（データオブジェクトの連続したフィールドの塊）の先頭アドレスをデータとして大小順にソートし，このソート済みデータを利用してヒープ中のクラスタのみを走査する．[2),3),10),11)] そのため，時間計算量は，

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

となる．nはクラスタの総数である．

一般的なアプリケーションでは使用中データオブジェクトの集まりはクラスタを形成しやすいので，nはAと比べて十分小さいことが予想される．そこで，式（1）は，複写型GCの時間計算量と同じ

$$O(A) \quad (2)$$

と近似できる．

また，両者は領域計算量（space complexity）においても同じである．機能的には，複写型GCの半領域（semi-space）が圧縮型GCの全領域に相当することから，複写型GCは記憶の使用効率が悪いと見なされている．しかし，このことは仮想記憶の計算機では問題とならないという説もある．

表1 高速圧縮型GC
Table 1 Fast Sliding Compaction GC.

GC名称	基本方式	ヒープ走査回数
LLGC[2)]	補正表（使用済領域内）	3
MOA[3)]	Morris法	2
佐藤・寺島法[4)]	補正表（未使用領域内）	2

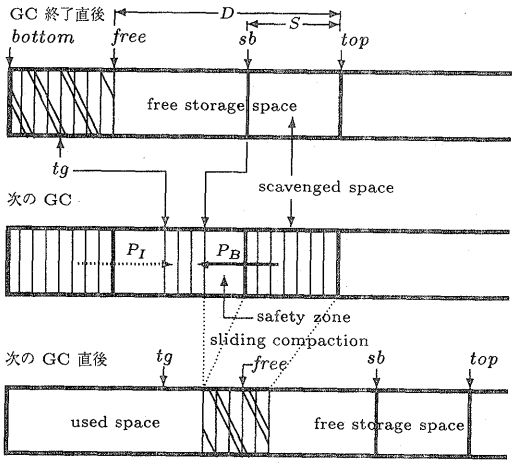
2.2 ポインタ補正

圧縮型GCでは使用中データオブジェクトの移動にともない，ポインタの補正が必要となる．ポインタ補正を高速に行う技法の1つに補正表[9)]を用いるものがある．補正表は，2^m $(m > 0)$フィールドごとに分割された格納領域（小区画）についてその先頭番地の補正値をエントリに持つ．ポインタはその参照先の小区画の代表値と小区画内のフィールド走査で求まる補正値の和をとることで補正できる．フィールドはデータオブジェクトの最小構成単位で，対象計算機のマシンワードに対応する．この技法は高速化を目的に次のような洗練が行われている[4),12)]．

(1) ヒープを2回だけ走査する．
最初の走査では，補正表を作成しながら同時に後向きポインタを補正する．後向きポインタとはその終点が始点より小さいアドレスであるポインタのことであり，その補正に必要な部分のエントリは作成済みである．しかも，同じクラスタを指すポインタならば，その補正値は即座に求まる．2回目の走査では，データオブジェクトの移動と前向きポインタの補正とが同時に行われる．

(2) エントリを効率的に使用する．
補正値を格納して余る各エントリの剰余ビットをその小区画の前半に含まれる使用済フィールドの数の記録に充て，その0値で小区画内の唯一のクラスタの補正値を表す．前者は小区画の4分割走査を実現する．m=5の場合には，走査フィールドは平均でわずか4である．後者は1小区画を超えるようなクラスタを指すポインタ補正をフィールドの走査無しで可能にする．

これらは，補正表による方式をMorris[13)]法に基づく方式よりもさらに優位にする[12)]．Morris法では補正表のような外部記憶を必要としないが，各フィールドに1ビット（Morrisの補助ビット）が要求される．また，データオブジェクトの共有が多い場合は，補正処理で補助ビットを使用してのポインタの入替えの操作が多用されるため，処理速度の低下を招くことが一般に知られている．



注 P_I：前向きポインタで終点が未回収領域を指す最大なもの．
P_B：後向きポインタで終点が未回収領域を指す最大なもの．

図 1 GC の動作
Fig. 1 Action of GC.

2.3 便宜的 GC

便宜的 GC はヒープがデータオブジェクトの使用量に対して十分に大きく確保できることを前提に考案されたものである．データオブジェクトはヒープのアドレスの小さい方から順に格納され，ヒープが一定量 D だけ消費されたときに GC が起動する．

この様子を示したのが図 1 である．*free* は新たにオブジェクトが格納される場所を示す．これは，オブジェクトが詰められた左側の領域と右側の空き領域（free storage space）との境界でもある．*top* は次の GC までに消費可能な空き領域の上限を指す．*free* が *top* を超える直前に GC が起動され，*sb* から *top* までが回収領域（scavenged space）となる．その量は S $(0 < S \leq D)$ である．ただし，これは見かけ上であり，これに隣接する非回収領域の一部を効率的に回収することも行う．この部分を安全地帯（safety zone）と呼ぶ．GC の終了に伴い，*free*，*sb*，*top* のそれぞれが更新され，新たなデータオブジェクトの格納のために空き領域が再設定される．

この回収領域は最も新しく作成されたデータオブジェクトが存在する領域であり，計算で使用中オブジェクトと，計算がきわめて短期間に終了して使用済となったデータオブジェクトが混在している可能性が高いと考えられる．それはまた，処理の高速化に直結する使用中オブジェクトの局所化の効果が最も得やすい場所でもある．こうした回収領域の限定化は世代別 GC[14)~16)] による限定化よりもさらに狭い範囲となる．その一方で，回収処理の対象とならなかった使用済オブジェクトが存在したり，使用領域が単調に増加するという問題点も生じる．

3. ハイブリッド GC

本論で述べるハイブリッド GC の目的は，2.3 章で述べた既成の便宜的 GC の問題点の克服にある．回収領域量（S）を少なくとれば，各回の GC 処理時間は短くて済む．しかし，非回収領域に残存する使用中オブジェクトは安全地帯の生成を阻み，結果としてヒープの使用量を増加させる．使用中オブジェクトが少なくなれば，ヒープの使用効率は明らかに低下する．

ハイブリッド GC は，ヒープの使用量を押さえるために非回収領域に残された使用中オブジェクトの圧縮処理を定期的かつ高速に行う機能をもつ．以下，これを Grand GC と呼ぶ．高速圧縮型 GC では，その処理時間は使用中オブジェクトの総量に比例し回収対象の領域量にはよらないので，使用中オブジェクトの量が少なければその負担増は問題とならないし，領域量でも大きな利得が得られることになる．また，remembered set[15)] の構造を単純化するために，Grand GC の回収領域は機械語コードが参照する Lisp オブジェクト（これらは *bottom* 以下のヒープ最下端に置かれる）を除いた使用領域全体となる．

3.1 アルゴリズム

Lisp 側に付加される前向きポインタの処理は次のように行われる．

1. 初期化 （処理系の開始時）
   $free \leftarrow bottom$; $top \leftarrow bottom + D$;
   $sb \leftarrow top - S$; $tg \leftarrow bottom - 1$;
2. **for each** 上書き操作 **do** {
   let 上書きフィールド **be** f;
   let 上書きポインタ **be** p;
   if $Adr(f) < sb$ **then**
   if $p > sb$ **then**
   $Adr(f)$ を remembered set に登録
   else if $Adr(f) < p$ && $tg < p$ **then**
   $tg \leftarrow p$}

ここで，$Adr(f)$ は f のアドレスを意味する．また，上書き操作はリストやブロックのフィールドを対象とするものに限られる．そこで，SET や PUT などのシンボルの上書き操作に伴う事後処理は GC 側が行うことになる．

GC 処理は，印付け，ソーティング，第 1 回ヒープ走査，補正，第 2 回ヒープ走査の順に進行する．印付けとソーティングに関わる圧縮型 GC の高速化技法や 2 回のヒープ走査を可能にしたポインタ補正法につい

ては他に詳細な文献[3),4),12)]もあるので，ここでは特に tg と sb の境界を跨ぐポインタの処理について述べる．なお，remembered set と GC の作業用スペースは top 以降のヒープが使われる．

1. GC の起動時
   $b \leftarrow tg$;
2. 印付け時
   for each f **in** オブジェクトフィールド
   do {
   let f の値（ポインタ）**be** p;
   if f はシンボルフィールド && $Adr(f) < sb$ && $p > sb$ **then**
   $Adr(f)$ を remembered set に登録
   if $p < sb$ && $b < p$ **then** $b \leftarrow p$;}
3. 境界調整（印付け終了後）
   if b+**sizeof** (b) $\geq sb$ **then** {
   comment promotion;
   $sb \leftarrow sb$+ 回収領域のアンカー☆の容量;
   for each e **in** remembered set **do**
   if e の値は前向きポインタ **then** {
   let e の値（ポインタ）**be** p;
   if $p < sb$ **then** {
   e を削除; **if** $tg < p$ **then** $tg \leftarrow p$;}}
   else e を削除;}
   else { **comment** 安全地帯の回収;
   $sb \leftarrow b$+ **sizeof**(b);
   for each e **in** remembered set **do**
   if not e の値は前向きポインタ **then**
   e を削除;}
4. 補正
   for each e **in** remembered set **do**
   if e の値はポインタ **then**
   e の値を補正;
5. 更新
   $top \leftarrow free + D$; $sb \leftarrow top - S$;
   if top >limit **then**
   comment 次回は Grand GC;
   $sb \leftarrow bottom$; $tg \leftarrow bottom-1$;}

GC は回収領域の使用中オブジェクトの印付け作業を行うが，このとき後向きポインタを調べ，その終点が非回収領域にある最大のもの（b）を求める．また，OBLIS 中のすべてのシンボルを調べて，それらのフィールドに sb を跨ぐ前向きポインタが存在すれば，それを remembered set に登録する．それらはポインタ補正の対象になる．次に b と sb を比べ，大きい方が指すオブジェクトの次が最終的な回収領域の下端となる．この地点と sb の間が安全地帯である．安全地帯ができれば，回収領域は既定の S よりも大きくなる．こうした現象は，データオブジェクトの寿命がきわめて短くて S がそれを完全に包括するような場合に現れる（図 3 参照）．top の更新が制限値（limit）を超える場合，次回の回収はそれまでの使用領域全体に拡張される．これが Grand GC である．

OBLIS 中のシンボルをすべて調べることは GC の負担増となる．その反面で，プログラムのコードに含まれる大域変数や special 変数に対する代入時検査が省かれ，結果的に remembered set の使用度や使用量も少なくなる．この方法を採用したのは，Lisp 側は負担減による利得が大きいと判断したためである．

3.2 処理の細分化

ハイブリッド GC のもう 1 つの利点は，GC 処理を時間的に細分化できることである．本章の冒頭で述べたように，回収領域量（S）を小さくすると各回の GC 処理時間を短くすることができる．これは GC 処理が時間軸で細分化できることを意味する．なお，非回収領域に残された散在する使用中オブジェクトはその後，定期的に行われる Grand GC がヒープ使用の効率化のために圧縮処理してくれる．一般的なプログラム[19)]で抜本的な処理速度の向上が望めるのは D が比較的大きい場合である．本 GC での $S/D \simeq 1$ は一世代についての世代別管理に相当するものである．

4. 実験環境

GC の性能評価は PHL 翻訳系の実行環境下で行うことで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ようにした．こうした環境では，プログラム自身や解釈系が生成するデータオブジェクト（Lisp の形式や連想リストなど）はヒープに存在せず，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．また，UNIX の OS 下では 1 つのプロセスが扱うことのできる記憶容量に強い制限があるため，多量のヒープを使用する大規模プログラムの実行には翻訳系が適している．

4.1 計算機環境

使用した計算機は，Sun Microsystems 社のワークステーション Enterprise 450（UltraSparc 300MHz）であり，一次キャッシュ容量は 32KB（命令 16KB, データ 16KB），二次キャッシュ容量は 2MB である．また，主記憶容量は 256MB（DIMM 144 ビット幅 60 nsec.）である．

☆ anchor，移動しない（再配置されない）クラスタのこと

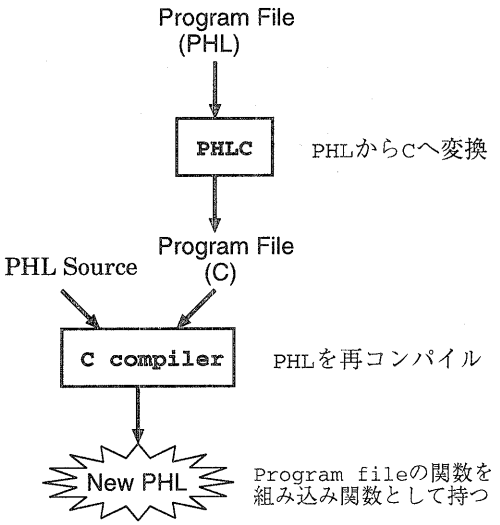


図 2 PHL 翻訳系
Fig. 2 PHL Compiler.

4.2 PHL 翻訳系

Lisp（PHL）から C 言語へのトランスレータ（PHLC）を新たに作成し，これと対象計算機の C コンパイラを結合することで翻訳系の環境を構築した．この翻訳系にハイブリッド GC の組込みを行った．PHLC は Lisp の関数と等価な C の関数をソースプログラムの形式で生成する．このため，生成コードに対する最適化（code optimization）やデバッグは C 言語のレベルで行うことができる．PHLC が生成するファイルは次の 4 個であり，a.out のような対象計算機のオブジェクトコードを直接に生成しない．

comp.c	関数定義
comp.h	同ヘッダファイル
comp.txt	Lisp フォーム（READ の処理対象）
init.c	C（の名前）と Lisp（のデータオブジェクト）の対応の処理

変換された関数は UNIX のコマンドである make を利用して新たな PHL に組込まれる．こうした方式の問題点の 1 つは，プログラム量の増加がファイル容量の増加に直結することである．たとえば，REDUCE[7] の基本パッケージを変換して作られる comp.c のファイルは約 22900 行（35445 tokens）となり，このコンパイルに gcc は CPU 時間で約 70 秒かかることになる．「翻訳即実行」という処理形態は PHL の変更なしには不可能であるが，翻訳するようなプログラムは一般に恒久的使用を目的に作られるので，こうした方式はパッケージを作るのに便利である．

PHLC が生成する C のソースプログラムは，peephole optimization などの局所的な最適化を行っているために比較的良質なものが得られている．ただし，最適化自体は本論文の主題ではないので，ここではこのことを述べるにとどめる．

5. 評 価

5.1 処理時間

本 GC の性能評価を 4 つのプログラムの実行結果で示す．TPU[17] と Boyer[18] はともに定理証明を行う Lisp プログラムである．TPU は比較的データオブジェクトの寿命が短く，Boyer は SETF などの副作用のある関数が多用されており，しかもデータオブジェクトの寿命が比較的長い．この両者は世代別管理の観点からは相反するプログラムである．他の 2 つは，数式処理システム REDUCE[7] 上で実行したプログラムである．**表 2** 中の REDUCE(1) は機能紹介用に添付された demo プログラム，REDUCE(2) は 5 行 5 列の逆行列を求めその検算を行うプログラムである．

まず，次回の GC までに消費できる領域量（D）を固定させて，それぞれについて回収領域量（S）を変化させたときの処理系全体と GC の処理時間を求めた．2 列表示の右側（Oc の表示欄）が便宜的 GC によるもので，左側（Hy の表示欄）がハイブリッド GC によるものである．括弧内はその内で Grand GC に要した時間である．なお，ハイブリッド GC はヒープ使用量を 150K 語（REDUCE(2) では 500K 語）に設定している．

便宜的 GC では，D に対する S の設定にも依るが，D が小さいと通常，ヒープの使用量も少なくなるため GC 処理を除いた純計算にかかる時間は短くなる．D が同一であるならば，S が大きいと回収効果が効いて純計算時間は短くなるが，逆に GC 処理時間は長くなる．反対に，S が小さいと GC 処理時間は一般に短くなる．GC の総回数は S によらず同じであるから，1 回当たりの GC 時間（Average GC time 欄）も短くなる．これが 3.2 章で述べた便宜的 GC による処理の時間細分である．

ハイブリッド GC では，各プログラムが生成したデータオブジェクトの寿命により処理時間に違いが生じる．寿命の短い TPU では便宜的 GC と同じ理想的な結果が得られている．Boyer では，長寿命オブジェクトの処理が GC の時間増を招き，それが処理時間全体の増加となっている．平均で約 35msec. を要する Grand GC が 1 回当たりの GC 処理時間を約 22msec. と，便宜的 GC の倍以上に押し上げているのである．これはハイブリッド GC がすべてに万能ではないこと

表 2 処理時間
Table 2 Processing Time.

TPU		Total time (sec.)		Total GC time (sec.)		Average GC time (sec.)		Total Heap (K words)		Compacted (K words)		Safety Zone (K words)	
D (K words)	S	Hy	Oc	Hy (Ggc)	Oc	Hy	Oc	Hy	Oc	Hy	Oc	Hy	Oc
500	—	0.15		0		0		449		0		0	
80	50	0.15	0.15	0.01 (0)	0.01	0.002	0.002	113	113	10.1	10.1	127	127
	30	*0.14*	*0.14*	0.01 (0.00)	0.01	0.002	0.002	183	232	9.0	8.5	53	105
	10	0.15	0.16	0.00 (0.00)	0.00	0.00	0.00	152	401	9.1	3.8	0.3	0.1
200	50		0.15		0.00		0.00		325		4.5		179
	30		0.15		0.00		0.00		371		4.0		172
	10		0.15		0.00		0.00		425		1.8		0.5
Boyer		Hy	Oc	Hy (Ggc)	Oc	Hy	Oc	Hy	Oc	Hy	Oc	Hy	Oc
500	—	0.20		0		0		397		0		0	
80	50	0.27	0.22	0.09 (0.07)	0.05	0.023	0.013	166	258	206	84.9	24	24
	30	0.26	0.23	0.09 (0.07)	0.03	0.023	0.008	199	336	183	64.1	4.0	5.2
	10	0.28	0.20	0.09 (0.08)	0.01	0.023	0.003	241	379	161	22.0	0.0	0.8
200	50		0.20		0.01		0.01		367		19.1		0.0
	30		0.20		0.01		0.01		382		14.5		0.0
	10		0.20		0.00		0.00		396		8.4		0.0
REDUCE(1)		Hy	Oc	Hy (Ggc)	Oc	Hy	Oc	Hy	Oc	Hy	Oc	Hy	Oc
500	—	0.38		0		0		466		0		0	
80	50	0.38	*0.37*	0.03 (0.01)	0.02	0.006	0.004	194	236	35.0	20.1	0.3	0.7
	30	0.38	*0.37*	0.03 (0.01)	0.02	0.006	0.004	188	331	29.5	16.2	1.1	1.2
	10	0.40	0.38	0.02 (0.02)	0.01	0.004	0.002	178	423	40.1	8.0	0.2	0.0
200	50		*0.36*		0.01		0.005		377		7.8		1.5
	30		*0.37*		0.01		0.005		411		6.5		0.8
	10		*0.36*		0.00		0.00		450		4.1		0.2
REDUCE(2)		Hy	Oc	Hy (Ggc)	Oc	Hy	Oc	Hy	Oc	Hy	Oc	Hy	Oc
2500	—	2.28		0		0		2478		0		0	
200	80	*2.26*	*2.24*	0.13 (0.08)	0.05	0.011	0.004	601	1502	121	48.5	57	64
	50	2.29	*2.24*	0.13 (0.11)	0.03	0.011	0.003	606	1710	150	40.0	160	214
	30	2.31	*2.25*	0.13 (0.11)	0.02	0.011	0.002	586	2026	157	34.7	82	126
400	80	2.30	*2.27*	0.13 (0.10)	0.03	0.022	0.005	773	2007	99	26.2	18	25
	50	2.30	*2.25*	0.11 (0.09)	0.02	0.018	0.003	786	2082	94	21.0	53	124
	30	2.32	*2.25*	0.10 (0.09)	0.01	0.017	0.002	817	2235	91	17.8	27	46

を示すものであるが，この種のプログラムを効果的に処理するには多世代管理の機能が必要となる．また，REDUCE(1) の結果は TPU に，REDUCE(2) の結果は Boyer に類似したものとなっている．なお，すべてのプログラムで GC を除く純計算時間の短縮化は実現されている．

この 2 種の GC によるヒープ使用量の節約はそれが対象計算機のキャッシュ記憶に留まる確率を高め，いわゆるキャッシュヒットの効果を生む．ヒープとスタックを含まない PHL の処理系部が約 600KB であり，二次キャッシュ容量が 2MB であることを考慮すると，300MB 程度のヒープならば確実に高速記憶に常駐される．これはハイブリッド GC で顕著に表れ，結果として純計算時間の短縮を生み出す．また，便宜的 GC では S が大きいときに同様の効果が生じる．こうした純計算側の利得が GC 側の損失よりも大であるとき，表 2 中の斜体が示すように処理時間全体の短縮が実現される．なお，GC を起動しない場合は，3 章で述べた純計算側の上書き操作に伴う前向きポインタの処理部をあらかじめ無効にして実行するので，この処理のオーバヘッドはない．

5.2 ヒープ使用の様子

図 3 は TPU プログラムを実行したときのヒープの使用状況である．縦軸が GC の起動回数，横軸がヒープの使用量を表す．横に引かれた実線や点線と黒丸が各回の GC が起動されたときのヒープの状況を視覚的に表している．点線は実線の部分も含めて GC の起動までに消費した部分である．実線は既定の回収領域である．黒丸は GC が使用中オブジェクトを圧縮した位置とその大きさで圧縮オブジェクトの総量を相対的に表す．近接する 3 つの線でそれぞれ回収領域量が異なる（下から，S=10，30，50K 語）別個の GC による効果を対比的に示している．

図 3 (a) は D=80K 語で便宜的 GC が起動される

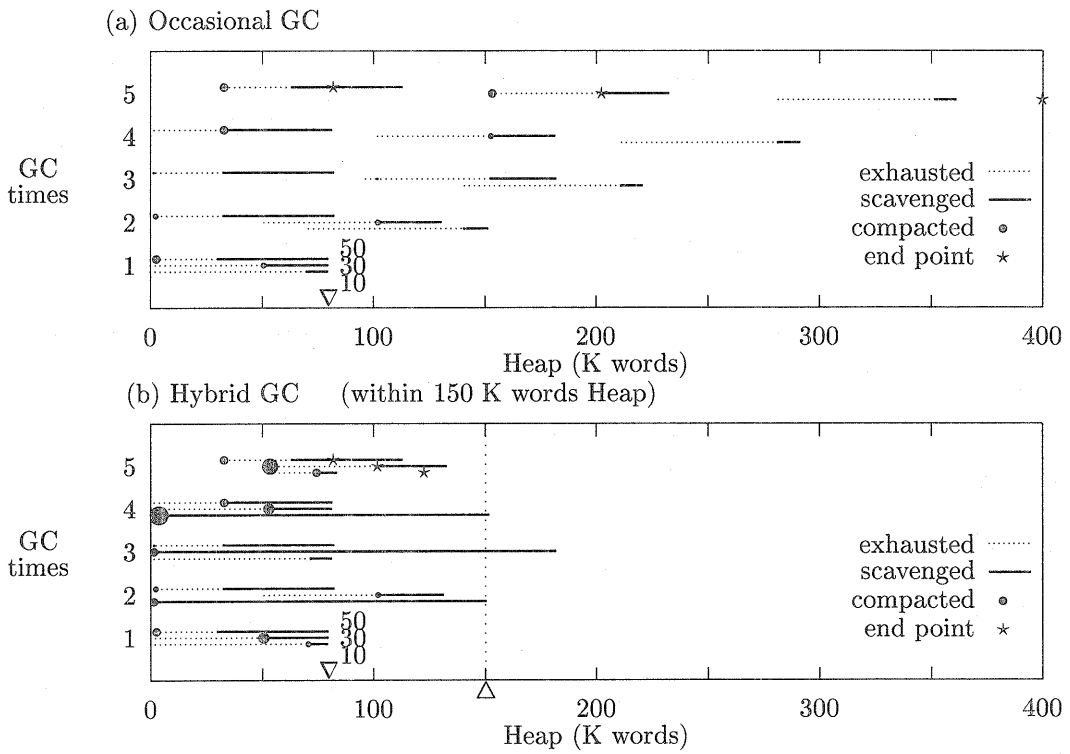


図 3 ヒープ使用の様子 (TPU)
Fig. 3 Utilization of a heap (TPU).

場合である．S が大きいと，回収領域より下位に圧縮が行われる場合がある．これが表 2 中の Safety zone 欄が示す安全地帯の容量で，S よりもこの分だけ回収量が大きくなる．各回の GC で圧縮位置が不動ならば，S を大きくしてもヒープ使用に変化はない．一方，非回収領域の上端に使用中データオブジェクトが存在すれば，回収領域の下端に圧縮が行われ，これ以上の回収量の増加はない．

図 3 (b) はヒープの使用制限を 150 K 語にしたハイブリッド GC の場合である．D は 80K 語で (a) と同じ条件である．この場合，横線は制限値を越えずに左側に戻る．ハイブリッド GC はそれまで未回収で残された領域を効果的に処理してヒープ使用の増加を抑止する．このことは，S の設定の大小に関係なくヒープ使用量がほぼ同じになることを意味する．そこではむしろ，S を比較的小さい値に設定することで各回の GC 処理時間の短縮とそれにともなう処理時間全体の短縮も図られるのである．これがハイブリッド GC の持つ大きな効果である．

6. おわりに

本論文では便宜的 GC の機能を拡張したハイブリッド GC が処理系にもたらす効果について，普遍的な実行環境と考えられる PHL 翻訳系で得られたデータに基づいて述べた．ワーキングセットを小さくすることでリスト処理系を高速にすることが GC の主目的である現在，使用中オブジェクトを局所化し，ヒープの使用量を一定に押さえる機能を持つハイブリッド GC はこれに大いに寄与するものと考える．

世代別管理が効果的に働くプログラムについては，本 GC はその回収領域の大小によらず効果的に処理できることを示した．回収領域の縮小は GC の処理を時系列で細分化することになり，比較的緩やかな実時間処理への適用が可能となると考える．

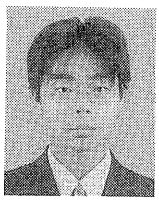
謝辞 本論文の先駆的研究で成果 [4),20),21)] をあげてくれた，電気通信大学大学院前期博士課程院生の佐藤圭史君および同大学情報工学科卒研生の亀井秀雄君と川田哲君の諸氏に感謝します．

参考文献

1) Wilson, P. R.: Uniprocessor Garbage Collection Techniques, Technical Report, University of Texas, Tex. (1994). (previous version

was published in *IWMM92*, Bekkers, Y. et al. (Eds.), LNCS 637, Springer-Verlag, pp. 1–42 (1992)).
2) Suzuki, M. and Terashima, M.: Time- and Space-Efficient Garbage Collection Based on Sliding Compaction, 情報処理学会論文誌, Vol. 36, No. 4, pp. 925–931 (1995).
3) Suzuki, M., Koide, H. and Terashima, M.: MOA – A Fast Sliding Compaction Scheme for a Large Storage Space, *IWMM95*, Baker, H.G. (Ed.), LNCS 986, Springer-Verlag, pp. 197–210 (1995).
4) 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 (1999) に掲載予定).
5) 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, 情報処理学会, pp. 17–24 (1995).
6) 佐藤圭史, 青木 徹, 寺島元章: Alpha-chip マシン上の PHL 処理系について, 電子情報通信学会技術報告 COMP97-94, pp. 57–64 (1998).
7) Hearn, A. C.: *REDUCE User's Manual, version 3.4*, The Rand Corporation, CA. (1988).
8) 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).
9) Terashima, M. and Goto, E.: Genetic order and compactifying garbage collectors, *Information Processing Letters*, Vol. 7, No. 1, pp. 27–32 (1978).
10) Sahlin, D.: Making Garbage Collection Independent of the Amount of Garbage, SICS Research Report R86008, SICS, Box 1263 S-163 13 Spånga, Sweden (1987).
11) Carlsson, S., Mattsson, C. and Bengtsson, M.: A Fast Expected-Time Compacting Garbage-Collection Algorithm, *ECOOP/OOPSLA'90 Workshop on Garbage Collection in Object-Oriented Systems* (1990).
12) Terashima, M., Ishida, M. and Nitta, H.: The Design and Analysis of the Fast Sliding Compaction Garbage Collection, *Advanced Lisp Technology*, Yuasa, T. et al. (Eds.), Gordon and Breach (published in 1999).
13) Morris, F. L.: Time- and Space-Efficient Garbage Collection Algorithm, *CACM*, Vol. 21, No. 8, pp. 662–665 (1978).
14) 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).
15) Unger, D. M.: Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm, *ACM Conference on Practical Programming Environments*, pp. 157–167 (1984).
16) Unger, D. and Jackson, F: An Adaptive Tenuring Policy for Generation Scavengers, *ACM Trans. on Programming Languages and Systems*, Vol. 14, No. 1, pp. 1–27 (1992).
17) Chang, C. L.: The Unit Proof and the Input Proof in Theorem Proving, *JACM*, Vol. 17, No. 4, pp. 698–707 (1970).
18) Gabriel, R. P.: Performance and Evaluation of Lisp Systems, Stanford University, CA. (1980).
19) Clinger, W. D. and Hansen, L. T.: Generational Garbage Collection and the Radioactive Decay Model, *ACM PLDI'97, SIGPLAN Notices*, Vol. 32, No. 5, pp. 97–108 (1997).
20) 亀井秀雄: 64ビット計算機でのリスト処理系の開発研究, 電気通信大学情報工学科卒業論文 (1999).
21) 川田 哲: PHL64 コンパイラ, 電気通信大学情報工学科卒業論文 (1999).

(平成 11 年 3 月 10 日受付)
(平成 11 年 4 月 28 日採録)



宮本 崇生

昭和 50 年生. 平成 10 年電気通信大学電気通信学部情報工学科卒業. 現在電気通信大学大学院情報システム学研究科博士前期課程に在学中. 記号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 年東京大学理学部物理学科卒業. 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 年より電気通信大学計算機科学科勤務. 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, 記号数式処理系などに興味を持つ. AAAI, ACM の各会員.