記 号 処 理 66－5
（1992. 9. 22）

Cベースのオブジェクトのための複写式回収器

小野寺 民也
日本アイ・ビー・エム株式会社　東京基礎研究所

C言語ベースのオブジェクトのための高性能の複写式回収器について述べる。それは慎重型でかつ世代型の回収器である。慎重型にするにあたっては、ポインタの大部分を正確に把握するために、コンパイラに支援を要請した。世代型にするにあたっては、記憶集合を管理するために、OSの提供する仮想記憶命令を利用することにした。実行時間特性は、自動回収は手動回収よりも効率が良くなる場合があることを示唆している。実際に性能を測定してみたところ、手動回収よりも高速になるという結果を得た。

Generational and Conservative Copying Collector for C-Based Objects

Tamiya Onodera

IBM Research, Tokyo Research Laboratory
5-19, Sanbancho, Chiyoda-ku, Tokyo, 102 Japan

A copying collector has two excellent properties, namely, that it compacts the heap and that the execution time depends solely on the number of live objects. Using a copying collector is expected to be a more efficient way of managing the heap than explicit freeing of objects. This paper describes a high-performance copying collector for C-based objects, which is both conservative and generational. We rely on the capabilities of the overlying compiler to identify most true pointers, and on support from the underlying operating system to detect pointers to younger generations.The performance results have actually confirmed our expectation; the collector has been faster than explicit freeing.

1 はじめに

不要になった記憶域 (garbage) を**自動回収**する方式で、現時点で有望視されているのが、**複写式** (copying) と**印掃式** (mark-and-sweep) である。双方とも、生きたオブジェクトの数に比例した実行時間でゴミを回収することができる。一方、C や C++のプログラマはふつう手動回収を行なっており、オブジェクトが最早使われていないことを自ら判断し陽にオブジェクトを解放する。陽に解放するには、少なくとも、ゴミとなったオブジェクトの数に比例する時間がかかる。このことは、自動回収を用いる方が効率がよい場合があることを示唆している。

本論文では、われわれが設計し実装した複写式回収器について述べる。実装は、NeXT Mach 2.0 上でオブジェクト指向言語 COB [13] に対して行なった。第2節でまず回収器の特徴について言及し、第3節で設計の要点について述べる。第4節では、回収器を含むヒープ管理システムによって使用されるデータ構造について述べ、割当器などの、回収器以外の部分の動作について説明する。第5節で回収器のアルゴリズムを詳説し、第6節で性能の測定結果を示す。第7節では関連研究を紹介し、最後に第8節で結論を述べる。

2 回収器の特徴

われわれの回収器は、複写式で**世代型** (generational) で**慎重型** (conservative) であるといった特徴をもつ。これら3つの修飾語をもつに至った理由を順に述べる。

複写式と印掃式の2つの方式に優劣をつけるのは相当微妙な問題であるが、われわれはヒープ領域を圧縮できる点を重視して複写式を選択することにした。もう少し詳しくいうと、実は、われわれはワーキングセットの大きなプログラムを効率よく動かすことに関心があった。このためには、仮想記憶のページング活動をある程度抑制することが肝腎で、ヒープ圧縮はこの抑制に大いに貢献すると考えたのである。

世代型回収 [9] [12] は、前節で述べた自動回収の実行時間特性を現実のものとするのに不可欠な考えである。自動回収が手動回収よりも優位な性能を示すためには、各回収におけるオブジェクトの生存率を低く抑える必要がある。世代型回収では、新たに生まれたオブジェクトの回収を主として行い、相対的に古いオブジェクトの回収を相対的に稀に行う。すなわち、たいていの回収において生き延びたオブジェクトの堆積は無視され、オブジェクトの生存率の漸増は回避されるのである。

慎重型回収 [6] [4] という考えは、プログラム言語や言語処理系を修正することなく自動回収を組み込むことを可能にする。慎重型回収器は、記憶域におけるポインタの場所についての正確な知識をもっている必要はない。すなわち、いわゆるタグあるいは実行時型情報の存在しない状況でもポインタを追跡しようとする。トリックは、ポインタであるか否か曖昧な語をすべてポインタと見做すことである。慎重型回収を用いずにスタック中のポインタを正確に追跡しようとするならば、言語処理系を改造してスタックフレームにおけるポインタの正確な場所を回収器に伝達する必要が生じる。この改造は単純な作業ではないし、そもそも高級言語がバックエンドになっている場合は不可能であろう。

最後に標題について補足すると、「C をベースにしたオブジェクト」は、具体的には COB を指しているのだが、「代入文の頻出するオブジェクト指向言語」および「実装上 C がバックエンドになっているオブジェクト指向言語」という2重の意味で用いられている。

3 回収器設計の要点

前節で示した特徴をもつ回収器を設計する際の、2つの要点について述べる。第1の要点は、いわゆる**記憶集合** (remembered set) をいかに管理するかということである。これは一般に世代型回収の設計における眼目である。記憶集合の要素、すなわち、あるオブジェクトの持つポインタで自分より若いオブジェクトを指しているものは、そのオブジェクトへの代入によってつくり出される。Lieberman らによる回収器 [12] は、特別な store 命令を使ってそのような代入を検知する。また、Appel の回収器 [2] は、すべての代入文をソフトウェアでチェックすることによりそのような代入を検知する。この回収器は、代入文の比較的稀な関数型言語 ML 用に作成されている。第3の方法は、仮想記憶命令を使うことである [3]。すなわち、古い世代のオブジェクトを含むページを書き込み禁止にすることにより、そのようなオブジェクトへの代入を捕捉するのである。世代型回収器を、特殊なハードウェアを用いずに、C ベースのオブジェクトのような代入文の頻出する言語に対して実装するには最善の方法だと考えられる。

第2の要点は、慎重型回収と複写式をいかに融合させるかということである。これらは基本的に相容れない。なぜなら、複写式回収器は、ポインタの値を更新するがゆえにポインタの場所を正確に知る必要があるからである。しかし、曖昧な領域をかなり限局できる

ならば、「曖昧な領域から参照されているオブジェクトは動かさず、それ以外の大部分のオブジェクトを複写移動する」という方針で、複写式回収を行なうことができる [4]。一般に、プロセスのメモリは、レジスタ、スタック、大域部、ヒープと4つの領域に分割して考えることができるが、以下のような支援をコンパイラから受けることにより、曖昧な領域をレジスタとスタックに限局することができる。すなわち、ヒープに割り当てられるオブジェクトに対して**ポインタ検出ルーチン** (pointer finding routines) を、大域部に割り当てられたポインタに対して**ポインタ登録ルーチン** (pointer registering routines) を、コンパイラに出力してもらうのである。このためにコンパイラを改造する必要があるが、作業量としては大きくない。回収器はこれらのルーチンを使って、ヒープおよび大域部にあるポインタに関しては正確に追跡することができる。

4 ヒープ領域管理システム

回収器の設計は割当器の設計と不可分であり、これらはいわゆるヒープ領域管理システム（以下、単に管理システムという）を構成している。本節の前半では、われわれの管理システムによって使用されるデータ構造について述べ、そのデータ構造が有する不変式 (invariants) を提示する。当然ながら、回収器もまたこれらのデータ構造を操作し、回収器の操作もまたこれらの不変式を保存する。回収器のアルゴリズムについては次節で述べることとし、本節の後半では、割当器およびトラップ処理器による操作について説明する。

4.1 データ構造

われわれの管理システムは、図1 に示すようなデータ構造に基づいている。ここで、n は世代数を、円は**記憶塊** (memory chunk) のリストを表すものとする。最初は、記憶塊リストはすべて空である。

管理システムは、OS から記憶塊を適宜獲得する。したがって、1つの記憶塊の大きさは OS のページの大きさの整数倍である。また、記憶塊リストは循環双方向リストとなっている。したがって、リスト全体を走査することなしに、ある記憶塊リストの全体を別の記憶塊リストに移したり、記憶塊リストからある記憶塊を削除したりすることができる。

プログラムの通常の実行中は、$r[i]$ $(1 \leq i \leq n)$ と $rw[i]$ $(0 \leq i \leq n)$ だけが使用されている。割当は $rw[0]$ においてのみ起こる。つまり、新しいオブジェクトがつくられる記憶塊は $rw[0]$ に属している。オブジェクトは、自動回収を生き延びて生存期間が長くなるにつれて、$rw[0]$ から $r[1] \cdots r[n]$ へと複写移動されていく。これをオブジェクトの**遷進** (promotion) という。$r[i]$ $(1 \leq i)$ の記憶塊は書き込み禁止になっており、代入が発生すると、書き込み禁止を解除されたうえで、$rw[i]$ $(1 \leq i)$ に移される。つまり、$rw[i]$ $(1 \leq i \leq n)$ は記憶集合を形成している。

自動回収の実行中では、後で詳述するように、$r\_ank[i]$ $(1 \leq i \leq n)$ と $rw\_ank[i]$ $(0 \leq i \leq n)$ は、曖昧な領域から参照されているために複写移動することのできないオブジェクトをもつ記憶塊を保持する。また、$r\_tos[i]$ $(1 \leq i \leq n)$ と $rw\_tos[i]$ $(0 \leq i \leq n)$ は、回収中にオブジェクトが複写されていく記憶塊を保持する。不要になり再利用可能になった記憶塊は自由リストへと移される。

世代0から世代 k までを対象とする自動回収を、$gc(k)$ で表すことにする。各世代 i は属性として時刻 $gct(i)$ を持っており、これは $gc(k)$ $(k \geq i)$ が起動されるだびに1つづつ増加する。一方、各記憶塊 c は属性として時刻 $gct(c)$ と世代 $gen(c)$ をもつ。記憶塊 c が記憶塊リスト $r[i]$ ないし $rw[i]$ に属しているとき、これらの属性は $gct(c) = gct(i)$ かつ $gen(c) = i$ という関係を満たす。したがって、記憶塊をある世代の記憶塊リストから別の世代の記憶塊リストに移した場合、この関係が維持されるように記憶塊の属性は更新される。

4.2 不変式

自動回収の実行中を除いて常に成立する不変式を、3つの群に分けて示す。あとでわかるように、第1群の不変式は複写式回収器の実行時間特性を維持するうえで重要な役割を果たす。

- 時刻と世代
  - 記憶塊 c が $r[i]$ または $rw[i]$ に属しているならば、$gct(c) = gct(i)$ かつ $gen(c) = i$ である。
  - 記憶塊 c が自由リストに属しているならば、$gct(c) < gct(gen(c))$ である。
- アクセス制御
  - 記憶域 c が $r[i]$ に属しているならば、c は読み出し可能だが書き込み禁止である。
  - 記憶塊 c が $rw[i]$ に属しているならば、c は読み出し可能で、かつ、書き込み可能である。
  - 記憶塊 c が自由リストに属しているならば、c は読み出し可能である。書き込み可能であるか否かについてはいずれでもよい。
- 参照関係

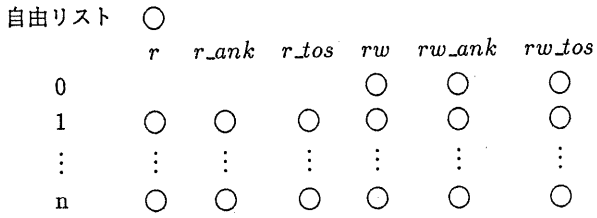

図1 ヒープ領域管理システムのデータ構造

記憶塊 c_1 に存するあるオブジェクトが記憶塊 c_2 に存するあるオブジェクトから直接参照されているとする。(以下では、これを単に「記憶塊 c_1 は記憶塊 c_2 から参照されている」ということにする。)このとき、

- 記憶塊 c_1 が $r[i]$ に属しているならば、記憶塊 c_2 は $r[j]$ $(1 \leq j \leq i)$ または $rw[j]$ $(0 \leq j \leq n)$ に属している。
- 記憶塊 c_1 が $rw[i]$ に属しているならば、記憶塊 c_2 は $r[j]$ $(1 \leq j \leq i)$ または $rw[j]$ $(0 \leq j \leq n)$ 上に属している。
- 記憶塊 c_2 が $r[i]$ に属しているならば記憶塊 c_1 は $r[j]$ $(i \leq j \leq n)$ または $rw[j]$ $(i \leq j \leq n)$ に属している。
- 記憶塊 c_2 が $rw[i]$ に属しているならば記憶塊 c_1 は $r[j]$ $(0 \leq j \leq n)$ または $rw[j]$ $(0 \leq j \leq n)$ に属している。

4.3 割当器とトラップ処理器

プログラムの通常の実行中に、管理システムのデータ構造は以下に示すように割当器とトラップ処理器により変更を加えられる。いずれの場合も不変式は成立し続ける。

- 割当器：新たな記憶塊が OS ないし自由リストから獲得され、$rw[0]$ に挿入される。
- トラップ処理器：プログラムが $r[i]$ にある記憶塊 c に書き込みを企てると、保護エラーが発生する。トラップ処理器はそのエラーを捕捉し、書き込み禁止を解除し、c を $rw[i]$ へと移す。実行制御がトラップ処理器を抜け出した後、違反した書き込みが再実行される。

割当器の動作についてもう少し詳しく述べる。各記憶塊は、連続した使用中の領域とそれに続く連続した未使用の領域からなる。この境界を管理するポインタを自由ポインタという。プログラムがある大きさの動的記憶を割当器に要求したとき、割当器は、$rw[0]$ の最初の記憶塊からその自由ポインタを進めることによって要求を満たそうとする。したがって、割当のコストは世代のない複写式回収器とほぼ同等であるということができる。最初の記憶塊が要求を満たすだけの未使用領域をもっていないとき、あるいは、$rw[0]$ が空であるとき、新たに記憶塊が自由リストないし OS より獲得され、$rw[0]$ の先頭に挿入される。

トラップ処理器の動作が示唆するように、記憶集合を管理するにあたって、われわれのシステムは、ページ保護を変更し保護エラーを捕捉しその処理をカスタマイズすることができなくてはならない。これらが可能であるか否かは、管理システムの基層である OS に依る。幸い、いくつかの OS はこうした仮想記憶命令をユーザプログラムに提供している。

5 回収器のアルゴリズム

本節では、回収器のアルゴリズムについて述べる。起動のタイミングについても言及する。

5.1 アルゴリズム

回収 $gc(k)$ $(0 \leq k \leq n)$ が起動されると、次の処理が順に実行される。

1. 世代 $i(0 \leq i \leq k)$ の時刻を 1 つ増やす。
2. スタックの内容を '慎重' に走査し、スタックから参照されている記憶塊を固定 (anchor) する。
3. ヒープ領域に形成されたオブジェクトのグラフを探訪 (traverse) し、複写移動可能なオブジェクトを複写移動する。
4. ゴミとなった記憶塊を自由リストに繋げる。
5. 生き残った記憶塊を遷進する。

第 2 から第 5 の処理を順に詳しく説明しよう。

スタック走査

スタック中の語をポインタだと見做したとき、$r[i]$ $(1 \leq i \leq k)$ ないし $rw[i]$ $(0 \leq i \leq k)$ に属する記憶塊 (の内部) を指しているならばその記憶塊を固定する。具体的には、記憶塊は $r\_ank[i]$ または

$rw\_ank[i]$ へと移される。固定記憶塊にあるオブジェクトはすべて生存しているとみなされ、かつ、複写移動されることはない。kより古い世代に属する記憶塊は無関係である点に注意されたい。

この第2の処理において、ある記憶塊をスタック中の語が指しているか否かを調べるのは、実は結構繁雑で、次の2つを順に確認する必要がある。

- スタックにある語が、管理システムによって管理されている記憶塊を指していること。
- その記憶塊が自由リストに属していないこと。

第1段階の確認が必要なのは、プロセスのヒープ領域全体がわれわれのシステムによって管理されているとは限らないからである。ある記憶塊がわれわれのヒープ管理システムによって管理されているものであるか否かを決定するために、印付け (memory marking) [8] というよく知られた技法を用いる。管理システムは、配列 $marks$ に、管理しているすべての記憶塊の番地を保存する。記憶塊が OS から獲得されたとき、管理システムは、配列 $marks$ の未使用の添字 i を選び、$marks[i]$ にその記憶塊の番地を格納し、記憶塊の最初の語に添字 i を格納する。したがって、スタック中の語 wに対して、$*mask(w)$ と $marks[*mask(w)]$ が等しいか否かを調べることにより、wが管理システムの記憶塊を指しているか否かが判定できる。ここで、$int\ *mask(void*w)$ は、語 wの下位のビットをクリアすることにより wの指すページの先頭のアドレスを整数へのポインタ型として返す†1。

ここで、さらに微妙な点が1つある。$*mask(w)$ という式が示すように、アドレス $mask(w)$ にある語は少なくとも読み出し可能でなければならない。$mask(w)$ がヒープの上限と下限の間にあることを確認するだけでは、その語が読み出し可能であるとはいえない。OS がヒープを連続して割り当てるとは限らないし、われわれの管理システム以外の例えばライブラリ・ルーチンによって管理されているヒープ領域は（稀だとは思うが）読み出し禁止となっているかもしれないからである。幸い、Mach は、あるアドレスが読み込み可能であるか否かを調べるシステム・コールを提供しているので、われわれはこれを用いることにした。

第2段階の確認について補足する。自由リストに属している記憶塊を固定集合に加えることは管理システムの暴走を招来してしまうので、この確認は必要な作業である。第4.2節の第1群の不変式は、まさにこの目的のために維持されている。

探訪と複写

第3の処理では、通常の複写式回収器と同じように探訪と複写が行なわれる。探訪の起点は、大域部のポインタ、記憶集合 $rw[i]$ $(k+1 \leq i \leq n)$ に属する記憶塊（i の範囲に注意）および、$r\_ank[i]$ $(1 \leq i \leq k)$ または $rw\_ank[i]$ $(0 \leq i \leq k)$ に属する記憶塊である。$r[i]$ $(1 \leq i \leq k)$ または $rw[i]$ $(0 \leq i \leq k)$ に属しているオブジェクトがポインタ p を通じて最初に探訪されたとき、オブジェクトは $r\_tos[i]$ または $rw\_tos[i]$ に複写され、p の内容が更新され、転送ポインタ (forwarding pointer) が元の場所に残される。元のオブジェクトがポインタ qを通じて再び探訪されたとき、回収器はそこに残されている転送ポインタの値で、qの内容を更新する。

スタックから参照され固定された記憶塊にあるオブジェクトはすべて生きていると見做されるので、これらは探訪の起点となる。また、これらのオブジェクトは複写移動の対象とはならない。

管理システムは、ポインタ登録ルーチンによって大域部にあるポインタの場所を正確に把握する。ポインタ登録ルーチンはソースファイルごとにコンパイラによって生成され、かつ、プログラムの起動時に ($main$ ルーチンが実行される直前に）自動的に呼び出されるように言語処理系が面倒をみる。呼び出されると、大域部にあるポインタの場所を管理システムに登録する。また、管理システムは、ヒープにあるオブジェクトのポインタの場所をポインタ検出ルーチンにより正確に同定する。ポインタ検出ルーチンはオブジェクトの型ごとにコンパイラによって生成される。COB では、すべてのオブジェクトの最初の語は、いわゆるメッソド表へのポインタになっており、このメソッド表の中にポインタ検出ルーチンのアドレスが埋め込まれている。回収器はオブジェクトの最初の語を通じてポインタ検出ルーチンを呼び出す†2。

ゴミの回収

第4の処理では、$r[i]$ $(1 \leq i \leq k)$ と $rw[i]$ $(0 \leq i \leq k)$ に存するすべての記憶塊が自由リストへと繋げられる。記憶塊リストは循環リストになっているので、自由リストに繋げるのに要する時間は回収の対象となった世代の数 kに依存し、移される記憶塊の数とは無関

†1 $mask(w)$ はページの先頭のアドレスではあるが、記憶塊の先頭のアドレスであるとは限らない。記憶塊が2ページ以上の場合もあるからである。この場合、ページ長だけ wを減らしながら判定を繰り返す必要がある。

†2 われわれの回収器は、多重継承が使われ導出ポインタ (derived pointers) が存在するときでも正しく動作する。基底ポインタと導出ポインタの差分がメソッド表に埋め込まれているからである。

係である。かくして、複写式回収器の実行時間特性が維持されている。

遷進

第5の処理では、$r\_ank[i]$, $r\_tos[i]$ $(1 \leq i \leq k)$, $rw\_tos[i]$ または $rw\_tos[i]$ $(0 \leq i \leq k)$ に属する記憶塊が、Figure 2に示すように遷進される †3。これらの記憶塊は、生き延びたオブジェクトを保持している。$(k+1)$ より古い世代は影響を受けない。

遷進する記憶塊の世代属性と時刻属性は、第4.1節で述べたような方法で更新される。記憶塊のリスト $rw\_ank[0]$ および $rw\_tos[0]$ は特別な扱いを受け、これらに属する記憶塊の保護は書き込み禁止に変えられる。

5.2 起動のタイミング

本小節では、いつ回収 $gc(k)$ を起動するのが最善かという問題について考える。前小節のアルゴリズムが示唆するように、回収が起動されるときのスタックの長さが回収器の性能に大きく影響する。スタックが長ければ長いほど、スタックの走査に時間がかかり、記憶塊が固定される可能性が大きくなる。われわれのアルゴリズムでは、スタックから記憶塊への参照が1つでもあれば、記憶塊にあるオブジェクト全体が生き残ると判断され、かつ、複写移動することができない。したがって、原則として、スタックが長いときは回収を回避するのが賢明である。

まず、$gc(0)$ の起動のタイミングについて述べる。$gc(0)$ の起動は、前回の回収以降ある一定数の記憶塊が割り当てられたときに試みられる。が、現在のスタック長があるスッタク閾値を越えているときは拒絶され実行されない。スタック閾値は、プログラムの最初のルーチン（Cでいう *main* 関数）に実行制御が移った時点のスタック長よりも少しだけ大きな値に初期設定されている。スタック閾値は、$gc(0)$ の起動が拒絶されるたびに、スタック閾値増分だけ大きくなる。このようにスタック閾値を動的に変えることにより、回収が起動されないままプログラムが長時間走行するのを避けることができる。ひとたび $gc(0)$ が起動されたなら、スタック閾値は初期値にリセットされる。一方、スタック閾値増分の値は現在の実装では動的に変わらない。スタック閾値増分の値が大きいほど、自動回収は起動されやすくなる。

スタックが十分短くなるときを最も良く知っているのはしばしばプログラマである [9]。たとえば、長時間走行するプログラムの多くはいわゆるトップレベルのループを持っており、繰り返し部の最後は自動回収の絶好の機会であると考えられる。われわれの管理システムは、回収の起動機会に関してプログラマと協調するために *appl_hint*() なるルーチンを提供している。このルーチンが呼ばれたとき、もしも $gc(0)$ 起動の拒絶の累積があれば、$gc(0)$ が直ちに起動される。

次に、$gc(k)$ $(k>0)$ の起動について述べる。現在の実装では、$gc(k)$ $(k>0)$ の起動方法は、$gc(0)$ のそれを単純に踏襲している。すなわち、一定数の $gc(k-1)$ の実行されたときに $gc(k)$ の起動が試みられ、ただし、スタック長がスタック閾値を越えている場合は拒絶されるようになっている。最近の研究 [11] によると、古い世代のオブジェクトの消滅パターンは若い世代のオブジェクトのそれと相似ではない。したがって、$gc(k)$ $(k>0)$ の起動方法については、別の方法を模索する必要があると考えている。

5.3 議論

遷進に際して記憶塊の属性を更新することにより、第1群の不変式が維持されている点に注目されたい。これらの不変式は、自由リストの記憶塊と使用中の記憶塊を区別するために維持されている。区別するだけでよいのであれば、このような不変式を維持するかわりに、たとえば、ゴミとなり自由リストにつながれる記憶塊の世代の値を-1といった特殊な値に再初期化する方が簡単であろう。しかし、これでは、第4の処理において、ゴミとなった記憶塊の数に比例した時間がかかることになり、複写式回収器の実行時間特性を喪失してしまうことになる。われわれのアルゴリズムのように、生き残り遷進する記憶塊の属性を変えて区別を創出しなくてはならないのである。

また、記憶塊の遷進に関しては、次のような方法も考えたがうまくいかないことがわかった。固定集合に属する記憶塊は、スタックから参照されているのですぐに書き込みが行なわれる可能性が高い。とすれば、$rw\_ank[0]$ にある記憶塊の保護を書き込み禁止にしてしまうのは効率が悪いであろう。したがって、これらを $rw[0]$ に再び繋ぎ戻し、$rw\_tos[0]$ にある記憶塊だけを $r[1]$ に遷進させる—という方法は一見合理的であるように思われる。しかし、これでは $r[1]$ から $rw[0]$ へのポインタが生起するかもしれず、第3群の不変式の破壊につながる。

†3 $k=n$ という特殊な場合は省略する。

	r	r_ank	r_tos	rw	rw_ank	rw_tos
0				-	-	-
1	$rw\_ank[0]$ + $rw\_tos[0]$	-	-	-	-	-
2	$r\_ank[1]$ + $r\_tos[1]$	-	-	$rw\_ank[1]$ + $rw\_tos[1]$	-	-
⋮	⋮	⋮	⋮	⋮	⋮	⋮
k	$r\_ank[k-1]$ + $r\_tos[k-1]$	-	-	$rw\_ank[k-1]$ + $rw\_tos[k-1]$	-	-
k+1	$r\_ank[k]$ + $r\_tos[k]$ + $r[k+1]$	-	-	$rw\_ank[k]$ + $rw\_tos[k]$ + $rw[k+1]$	-	-

図2　記憶塊の遷進：プラス演算子は2つのリストの連結を表し、マイナス記号は記憶塊リストが空であることを示している。

プログラム	実行時間 (sec)	生成数 (M 個)	回収時間 (sec)	起動回数		生存率 (%)	陽に解放 (sec)
				$gc(0)$	$gc(1)$		
割当ループ	92.8	2.50	9.0	395	23	0.27	235.9
Lisp インタプリタ	1244.9	1.06	5.7	29	3	1.05	114.9
COB コンパイルサーバ	1612.2	5.49	500.9	150	18	3.98	541.5

表1　測定結果

6　性能評価

本節では、NeXT Mach 2.0 でプログラム言語 COB 用に実装した回収器の性能について述べる。プロセッサは 25MHz のモトローラ 68030 であり、実メモリは 16 MB である。

3つのプログラムに、作成した回収器を組み込んで実行時間などを測定した。それらは、単純な割り当てループ、Lisp インタプリタ、COB コンパイルサーバで、すべて COB で書かれている。第1のプログラムは、Boehm らの論文 [7] で使われているものと同じもので、8バイトのオブジェクトを 50 万個次々と割り当てては（回収されるように）参照を消していく。Lisp インタプリタでは、300 以下の素数を求め逆に整列することを 32 回繰り返した。最後の COB コンパイルサーバ [14] は、ずっと複雑で現実的なプログラムである。今回の測定では、コンパイルサーバ自身のソースファイル 143 個を一気にコンパイルした。

NeXT Mach 2.0 のページ長は 8KB である。回収器のパラメタを調節して、記憶塊が 8 個消費されるたびに $gc(0)$ の起動が試みられ、$gc(0)$ が 8 回起動されるたびに $gc(1)$ の起動が試みられ、$gc(k)$ $(2 \leq k)$ は起動されないようにした。スタック閾値増分は 16 バイトに設定した。さらに、Lisp インタプリタと COB コンパイルサーバでは、関数 $appl\_hint()$ の呼び出しを1つずつプログラム中の適当な場所に挿入した。かくして、望ましくない時期に回収が起動されることはほとんどないようになっている。

測定結果を表1に示す。COB コンパイルサーバを例に説明すると、実行が終了するまでに 1612.2 秒かかり、549 万個もの大量のオブジェクトが作られた。自動回収のオーバヘッドは 500.9 秒で、$gc(0)$ は 150 回、$gc(1)$ は 18 回起動された。オブジェクトの生存率は、記憶塊の生存率で近似すると 3.98%であった。3つのプログラムは同一の値のパラメタで測定された。Lisp

インタプリタの場合、基本的に再帰を多量に伴うためスタックは長くなる傾向がある。したがって、実際に起動される自動回収の数は少なくなっている。

比較のため、自動回収のかわりにソースプログラムに関数 *free*() の呼び出しを注意深く挿入した場合のオーバヘッドを見積もってみた。最後の列に示されている。これらの数字は、割当に要する時間、すなわち、関数 *malloc*() で費やされる時間は含んでいない。また、見積もりにあたっては、時間測定用関数を呼び出し実行するオーバヘッドが過剰にならないように注意した。いずれの場合も、回収器を用いたほうが効率がよいことがわかる。表 1の生存率を考えてみれば、これは別に驚くに値しない。Appel も指摘しているように [1]、「自動回収を用いればプログラミングは簡単になるけれども、プログラムは遅くなる。」というのは真実とは限らないのである。

表2は、中断時間を示している。割当ループと Lisp インタプリタでは中断時間は十分短いといえるが、COB コンパイルサーバではそうでない。自動回収を対話型のプログラムに組み込む場合には中断時間が重要になるが、たとえば、自動回収を用いるように C 系プログラマを説得する場合には回収時間のほうがより重要になると思われる。

表3と表4は、コンパイルサーバの実行においてスタック閾値増分の値を変えたときの測定結果を示している。表3により、増分の値が大きいほど、起動を許される回収の数が増えることがわかる。回収時間もそれに応じて増加し、全実行時間もほぼその分だけ増加する。また、増分の値が大きいほど、スタックが長い状態で回収が起動されやすくなり、固定される記憶塊の数が増えることもわかる。一方、表4は、適度に頻繁に回収を起動すると平均中断時間も最大中断時間も改善することができることを示している。一般に、回収時間と中断時間はトレードオフの関係にあるといえる。

7 関連研究

慎重型の複写式回収器は、最初に Bartlett [4] によってプログラム言語 Scheme 用に提案された。慎重型にするという点に関しては、われわれも同じように "複写できないものは複写しない" というトリックを用いている。

C や C++といったシステムプログラム向けの言語のための回収器で複写式であるものは少ない。Edelson と Pohl [10] による C++用のものがあるがこれは慎重型でもなければ世代型でもない。スタック上のポインタをも正確に把握しようとしているため、たとえば、ポインタを引数としてもつ関数が呼び出されるたびに、余分なオーバヘッドがかかってしまう。しかも、現在の実装方式が複雑なコーディング・プラクティスに依存しているため、単純な C++プログラムでの実験結果しか示されていない。

Bartlett は慎重型の複写式回収器をさらに世代型とし、C++に組み込むことを提案している [5]。しかし、提案された回収器を C++ プログラムに組み込んだ場合の性能評価がなく、また、アルゴリズムが十分な詳しさで提示されていない。第5.3節で示したように、世代型にするにあたっては、複写式回収器の実行時間特性の喪失につながる落し穴があり、これを回避するためにちょっとした工夫が要るのである。

8 まとめ

コンパイラと OS の双方から適当な支援を得られるならば、C 言語ベースのオブジェクトに対しても高性能の複写式回収器を作成することができる。性能評価の結果が示すように、オブジェクトの生存率が十分低いとき、われわれの回収器を使用するほうが陽にゴミとなったオブジェクトを解放するよりも効率がよい。オブジェクト指向言語で素直に書かれたプログラムで長時間走行するものは、大抵、そのような生存率を示すものと考える。

COB コンパイラの修正が必要だったが、コンパイラによって生成されなければならないルーチンは、自動回収のためだけに有益なのではない。これらのルーチンは、オブジェクトを深く複写したり (deep copy)、オブジェクトを持続 (persistent) 化したり、オブジェクトを遠隔手続き呼び出し (RPC) で転送したりするのにも使用することができるのである。

謝辞

日頃より御指導いただいている上村務氏および久世和資氏に感謝いたします。また、小林真氏と大沢暁氏には Mach の使用に関してお世話になりました。

参考文献

[1] Appel, A.W. Garbage Collection Can Be Faster than Stack Allocation. *Information Processing Letters* 25,4(1987), 275-279.

[2] Appel, A.W. Simple Generational Garbage Collection and Fast Allocation. *Software Practice and Experience* 19,2 (February 1989), 171-183.

[3] Appel, A.W. and K. Li. Virtual Memory Primitives for User Programs. *Proceedings of the Fourth Interna-*

プログラム	*gc*(0)		*gc*(1)	
	最大 (msec)	平均 (msec)	最大 (msec)	平均 (msec)
割当ループ	40	23	29	18
Lisp インタプリタ	254	179	245	185
COB コンパイルサーバ	7560	2800	11790	4490

表 2　中断時間

増分 (bytes)	全実行時間 (sec)	回収時間 (sec)	起動回数		1 回収あたりの被固定記憶塊の数
			gc(0)	*gc*(1)	
16	1612	501	150	18	0.51
64	2136	1007	327	40	1.69
256	2733	1579	682	85	1.73
1024	3252	2105	1052	131	1.74

表 3　増分を変えたときの効果 (全実行時間、回収時間など)

増分 (bytes)	*gc*(0)		*gc*(1)	
	最大 (sec)	平均 (sec)	最大 (sec)	平均 (sec)
16	7.56	2.80	11.79	4.49
64	6.65	2.57	12.57	4.18
256	7.10	1.94	5.01	3.02
1024	10.30	1.66	5.95	2.72

表 4　増分を変えたときの効果 (中断時間)

tional Conference of Architectural Support for Programming Languages and Operating Systems (1991), 96-107.

[4] Bartlett, J.F. Compacting Garbage Collection with Ambiguous Roots. DEC WRL Research Report 88/2 (February 1988).

[5] Bartlett, J.F. Mostly-Copying Garbage Collection Picks Up Generations and C++. DEC WRL Technical Note TN-12 (October 1989).

[6] Boehm, H.J. and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience* 18,9 (September 1988), 807-820.

[7] Boehm, H.J. et al. Mostly Parallel Garbage Collection. *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (1991), 157-164.

[8] Comer, D. *Operating System Design, the Xinu Approach.* Prentice Hall, 1984, p.231.

[9] Demers, A. et al. Combining Generational and Conservative Garbage Collection: Framework and Implementations. *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages* (1990), 261-269.

[10] Edelson, D.R. and I. Pohl. A Copying Collector for C++. *USENIX C++ Conference Proceedings* (1991), 85-102.

[11] Hayes, B. Using Key Object Opportunism to Collect Old Objects. *OOPSLA'91 Conference Proceedings* (1991), 85-102.

[12] Lieberman, H. and C. Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM* 26,6 (1983), 419-429.

[13] Onodera, T. and T. Kamimura. *COB Language Manual.* IBM Research, Tokyo Research Laboratory (March 1990).

[14] Onodera, T. Reducing Compilation Time by a Compilation Server. Research Report RT0075, IBM Research, Tokyo Research Laboratory (April 1992).

[15] Ungar, D. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. *Proceedings of the ACM Symposium on Practical Software Development Environments* (1984), 157-167.