マークスイープとコピーの混合による効率的なゴミ集め

小林 義徳 †　　遠藤 敏夫 ††　　田浦 健次朗 †　　米澤 明憲 †

† 東京大学大学院情報理工学系研究科
†† 科学技術振興事業団

概 要

本稿では、*Rarely Copying Garbage Collector* (RCGC) というゴミ集め手法を提案、実装し、性能評価を行う。

RCGC の特徴はマーク後、各ページごとにコピーとスイープの有利な方を選択し、全体として高速な動的メモリ管理を可能にしていることである。さらに、一部のワードがポインタか否か不明な場合にも対応している。また、実マシン上でのベンチマークを通して、本手法の有効性を示す。

1　はじめに

本稿では、*Rarely Copying Garbage Collector* (RCGC) というゴミ集め手法を提案、実装し、性能評価を行う。RCGC は、高速なアロケーションを実現しつつ、領域ごとにコピーとスイープ有利な方を適用することにより高速なメモリ管理を可能にしていることと、曖昧なポインタ (実行時にポインタかどうか不明なワード) が一部存在する条件下でも GC が可能な保守的 GC であることが特徴である。

RCGC は曖昧なポインタに対応しているため、RCGC を用いた高級言語で書かれたプログラムに C 言語で書かれたモジュールを追加することが非常に容易であるという利点がある。また、RCGC は C 言語にコンパイルされるような高級言語用の GC としても有効である。

本稿の構成は以下のようになっている。第 2 節で RCGC の特徴について述べ、第 3 節で RCGC の方式について説明する。また、第 4 節でベンチマークを通して性能の評価を行い、第 5 節で、RCGC のパフォーマンスを最適にするパラメータについて考察・実験を行う。第 6 節で関連研究について述べ、第 7 節でまとめ、今後の課題について述べる。

2　Rarely Copying Garbage Collector の特徴

2.1　高性能な動的メモリ管理

高級言語での高性能の実現には、高性能な動的メモリ管理、つまり高速なアロケーション (メモリ確保) と高速な GC が不可欠である。

プログラムによっては、アロケーションの回数が一秒間に 10 万回を越えるものも珍しくなく、そのようなプログラムでは高速なアロケーションが特に重要である。

RCGC は高い頻度でリニアアロケーション方式を行うことにより高速アロケーションを実現する。リニアアロケーションは広大な空き領域の端から順にアロケーションする方法である。2 に示すように、アロケーションのコストは高々数マシン命令であり、フリーリストからのアロケーションに比べて高速である。また、リニアアロケーションは、アロケーション用のポインタ (図 2 での `free_ptr`, `limit_ptr`) をグローバルレジスタに取ることで、高速化が可能である。

既存の多くの言語処理系はリニアアロケーション用の広大な空き領域を得るために、コピー GC を採用している。しかしコピー GC におけるオブジェクト毎のコピーコストは、マークス

```
char *free_ptr, *limit_ptr;

void* gc_fast_alloc(size_t size)
{
    void* prev;

    prev = (void*)free_ptr;
    free_ptr += size;

    if(free_ptr < limit_ptr){
        return prev;
    }else{
         ...
    }
}
```

図 1: リニアアロケーション

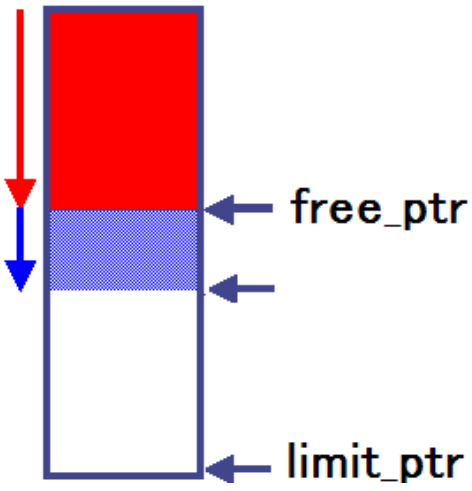

図 2: リニアアロケーション

イープ GC におけるマークコストより大きい。生きたオブジェクト量が多い場合には、多くのオブジェクトをコピーして少しの空き領域しかできない。このとき、コピー GC の合計メモリ管理コストはマークスイープ GC の場合より多くなる。

RCGC では、リニアアロケーションの実現のためのコピーをする一方、生きたオブジェクト量が多い場合にはコピーを避けることを基本方針にしている。このために RCGC はヒープを固定サイズのページに分割し、各ページに対してコピー、スイープのうち有利な戦略を適用する。まず、生きたオブジェクトのマークを行うと同時に各ページ中の生きたオブジェクト量を記録しておく。第二フェーズで (1) コピーか (2) スイープのいずれか有利と思われる方を行う。

2.2 曖昧なポインタへの対応

曖昧なポインタとは、実行時にポインタか否か不明なワードのことを指す。一般に、C 言語で用いられるメモリ上のワードは曖昧なポインタである。GC をサポートした高級言語では、処理系内部にどのワードがポインタかを区別するためのデータ構造を持っているのが一般的である。

GC つき高級言語で書かれたプログラムに C 言語で書いたモジュールを追加したい場合が多く存在する。また、高級言語をコンパイルする際に中間言語として C 言語を使いたい場合が多くある。たとえば、高級言語側の GC が曖昧なポインタに対応していない「正確な GC」の場合、C 言語でモジュールを書く際に、プログラマが全てのワードの型情報 (ポインタか否か) を明示しなければならない。これはプログラマにとっては負担であり、場合によってはバグの原因にもなりうる。また、言語処理系を作る際、そもそも言語処理系の実装に強く依存した「正確な GC」を書くこと自体大変な手間である。

RCGC は、言語処理系への組み込みが比較的容易で、なおかつ高速なアロケーション、小さい合計コストを目指した GC ライブラリである。RCGC は、Mostly-copying collector [2] と

同様、大部分のオブジェクトの型が分かっており、一部のオブジェクトの型がわからない、という状況を想定している。

他には、言語処理系に GC を組み込む場合、Bohem による保守的マークスイープ GC[4] を使うという選択肢もある。これは、全てのワードを曖昧なポインタとして扱うマークスイープ GC である。これを使うことにより、(1) 言語処理系を作る際に GC を実装する手間、(2) 高級言語で書かれたプログラムに C 言語のモジュールを追加する際の型を指定する手間、を省くことができる。しかし、この方法は、マークスイープ方式でありアロケーションはフリーリストより行われるため、アロケーション速度に問題がある。

3 Rarely-copying collection アルゴリズム

3.1 アロケーション方式

ヒープ全体は、等しいサイズのページ単位に分割されて管理される。ページは、以下のように分類される。

1. 丸ごと空き領域であり、将来リニアアロケーションに使われるページ
2. 現在リニアアロケーションに使われているページ
3. 生存オブジェクトが断片化して存在し、ページ内の空き領域がフリーリストで管理されているページ

3 のページに関し、断片化した空き領域は、フリーリストで管理される。フリーリストを探索せずに $O(1)$ でのコストでアロケーションを実現するため、フリーリストは空き領域のサイズごとに用意されている。具体的には、例えば 16 バイト以上 32 バイト未満の領域専用のフリーリストが存在し、各エントリーは複数ページにまたがって存在する。他に、32, 64, 128, ..., 4096 バイト以上用のフリーリストがそれぞれ存在する。

アロケーションは、以下のように行う。

1. リニアアロケーションに使われているページより、アロケート
2. リニアアロケーション用の領域が無くなったら、フリーリストよりアロケート。この時、まず要求されたサイズ用のフリーリストを使用する。なければ、より大きなサイズ用のフリーリストからアロケートする。
3. 要求されたサイズのオブジェクトがフリーリストに無い場合、丸ごと空きのページを新たにリニアアロケーション用に確保し、1 に戻る
4. ヒープのある割合以上のページが使われた時点で、GC を起動 (3.2 節で述べる)。ヒープのうち、GC が起こるまで使われなかった領域に、オブジェクトがコピーされる。

断片化している領域を先に使い切り、GC の起動タイミングを遅らせるため、空きページよりフリーリストを先に使うようにしている。

3.2 GC 方式

ページ内の生きているオブジェクトの量が多い場合、コピー量が多いわりに得られる空き領域が少なく、コピーするのは不利であり、スイープを選択した方が良い。逆に、生きているオブジェクトが少ない場合は、少ないコピーでページ全体を連続な空き領域として得ることができ、リニアアロケーションに利用できるため、コピーの方が有利である。従って、各ページに対し、生きているオブジェクトが少ない場合にコピー、多い場合にスイープを適用するのが望ましい。

Rarely-copying collector では、コピーかスイープか選択する基準として、閾値 C_{th} をあらかじめ設けておく。ページ内の生存オブジェクト量 L がマークフェイズで計算され、$L \leq C_{th}$ のページにはコピーを、$L > C_{th}$ であるページにはスイープを適用する。

また、GC 時にオブジェクトをコピーする場合、そのオブジェクトを指しているポインタも更新しなければならない。各ページごとに ER

set (Evacuated References set) [3] と呼ばれるデータ構造を用意し、マーク時に、ページ内のオブジェクトを指すポインタの位置が全て記録される。コピー時には、ER set の全エントリーに対し更新が行われる。ただし、マーク中にページ内の生存オブジェクト量が C_{th} を上回った場合、そのページはスイープされるため、それ以上 ER set への登録は行わない。

Mostly-copying collector[2] や 保守的マークスイープ GC [4] と同様に、レジスタ、スタック、大域変数をルートセットとし、そのなかのワードは全て曖昧なポインタとして扱う。それ以外のデータについては、プログラマが型を指定する。もちろん、型を指定せずに曖昧なポインタとして扱うことも可能である。

RCGC は以下のフェイズよりなる。

1. 初期化フェイズ
   GC に先立ち、マークスタック、フリーリストのクリアを行う。
2. マークフェイズ
   ルートセットよりポインタをたどり、生きているオブジェクトのマークを行う。同時に、各ページごとに、生存オブジェクトの量および数を記録しておく。また、ページ内のオブジェクトを指すポインタの場所を ER set へ登録する。
3. コピー or スイープフェイズ
   各ページに対し、コピーかスイープのどちらかを適用する。まず、ページ内のオブジェクトが曖昧なポインタに指されている場合、そのページにはスイープが適用される。そうでないページに対しては、マークフェイズで取得した情報に基づき、各ページごとにコピーかスイープの、有利と思われる方を選択実行する。具体的には、ページ内のオブジェクト生存量 L に対し、$L \leq C_{th}$ の場合にコピーを、$L > C_{th}$ の場合にスイープを適用する。
   ただし、生存オブジェクト量がゼロのページについては、丸ごと空き領域であるとして処理する。

   - コピーされるページ
     ページの、ER set の各エントリーに対し、指されているオブジェクトをコピー予約領域[1]にコピーする。同時に、オブジェクトを指しているポインタの更新を行う。コピー元の、オブジェクトがあった場所には、新しいオブジェクトを指すポインタが残される。コピー元のページは、GC 終了後、丸ごと空いているページとして扱われ、将来リニアアロケーションに使用される。
   - スイープ されるページ
     ページ内の、生きているオブジェクトのマークビットをクリアする。また、空き領域は、サイズごとに用意されたフリーリストに追加される。その際、隣接した空き領域は一つにまとめられる。

GC が起動されるのは、コピーするかもしれない最大量と、ヒープ内の空き領域 (コピー予約領域) の大きさが等しくなった時点である。これは、次式で表される。

$$A = \frac{M}{1+\beta} \tag{1}$$

ここに、A はアロケートされたオブジェクトの合計サイズ、M はヒープサイズ、$\beta = \frac{C_{th}}{\text{PAGESIZE}}$ $(0 \leq \beta \leq 1)$ は、コピーかスイープかの選択に用いられる閾値に対応する。とくに、$\beta = 0$ の場合、RCGC は完全なマークスイープ GC と同等である。また、$\beta = 1$ の場合は、曖昧なポインタに指されているページが「ピンダウン」[2]されることを除いては、RCGC はマークコピー GC と同等の動作をする。

[1] RCGC は、オブジェクトのコピーを伴うため、ヒープを全て使いきる前に GC が起動される。GC が起こるまで使われなかった領域が、コピー予約領域であり、GC 時に一部の生存オブジェクトがこの領域にコピーされる。

[2] 曖昧なポインタに指されているオブジェクトは安全に動かすことができないため、そのようなオブジェクトがあるページにはコピーではなくスイープが必ず適用される

3.3 実装の現状

Rarely-copying collector は、現状では UNIX 上でのみ動作する。ソースは、C 言語で約 5700 行である (内、ルートセット取得部分は Boehm GC からの流用であり、約 1300 行)。現在のところ、Linux(i386), SunOS(Sparc) での動作を確認している。Sparc ではグローバルレジスタを使ったリニアアロケーションも行うことができる。閾値 C_{th} の設定については、GC 実行前にあらかじめ単一の閾値を定めておく方式のみ実装してある。

4 評価

4.1 アロケーション速度の比較

ひたすら同じサイズのオブジェクトのアロケーションを行うだけのプログラム (図 3) を用い、保守的マークスイープ GC(Boehm GC) と RCGC で実行時間を比較したものを、図 1 に示す。RCGC のアロケーションは 3.1 節で述べたように、1 ページをリニアアロケーションに使い、無くなったら新しいページを確保するという方式である。

また、Boehm GC のアロケーション方式は、基本的にフリーリストからのアロケーションである。特定のサイズ専用のページおよびフリーリストが用意されており、そこからアロケーションを行う。このため、フリーリストの無駄な探索が行われることがなく、$O(1)$ の時間でアロケーションが完了する。

比較にあたり、マシンはSUN Enterprise 4500 が用いられた。CPU は UltraSPARC II 400MHz, OS は SunOS 5.8 である。また、RCGC については、リニアアロケーション用のポインタは、グローバルレジスタ上に確保されており (2.1 節を参照)、全てのアロケーションはインライン展開されている。実験によると、最もアロケーションが頻繁に起こるプログラムでは、実行時間が約 4 倍異なることがわかる。

```
GC_INIT(heapsize);

gettimeofday(&tv1,NULL);
for(i=0;i<iteration;i++){
    MALLOC(objsize);
}
gettimeofday(&tv2,NULL);
```

図 3: アロケーション速度の比較に用いたプログラム

objsize = 40, iteration = 10000000
heapsize = 1024*8192 bytes

使用した GC	実行時間	GC 回数	GC 時間
Boehm	3.993	58	0.103
RCGC (C_{th} = 2048)	0.973	72	0.104

表 1: アロケーション速度の比較

4.2 メモリ管理全体としての性能評価

RCGC の性能は、コピーかスイープかを決める閾値 (C_{th}) により、大きく左右される。そこで、いくつかのプログラムを用い、戦略選択に用いる閾値 (C_{th}) を変えながら、それぞれの閾値に対する実行時間を測定した。ベンチマークには以下のプログラムが用いられ、SUN Enterprise 4500 上で行われた。CPU は UltraSPARC II 400MHz, OS は SunOS 5.8 である。

それぞれヒープサイズは固定で、ER set は十分大きくとって実験を行った。

deltablue

制約グラフを入力とする制約解消プログラム。

N-body

Barnes-Hut アルゴリズム [1] により N 個の質点の運動をシミュレートする。

cube

ルービックキューブを解くプログラム。

グラフの横軸はいずれも閾値を表す。図4、図7および図10は、プログラムの実行時間を表す。ただし、**N-body**, **cube** に関しては、実行時間の原点が 0 でないことに注意されたい。実行時間のグラフで`app+fastalloc`, `total_slowalloc` 以外の部分の合計が GC 時間を表す。実行時間のうち主要なものは、グラフ中で下から順に、アプリケーション実行時間 + リニアアロケーション時間 (`app+fastalloc`), リニアアロケーション以外のアロケーション時間 (`total_slowalloc`), スイープ時間 (`total_sweep`), マーク時間 (`total_ma`
コピー時間 (`total_copy`) である。

また、図5 や 図8、図11 は、GC 時にスイープされたページ数、空だったページ数、コピーされたページ数を表す。また、図 6、図 9、図 12 は、各 GC 時に生き残ったオブジェクトの、プログラム実行全体を通しての総和を表す。

また、一部の結果を表 2 から表 4 にまとめた。表中で Boehm とあるのは、RCGC ではなく、Boehm らによる保守的マークスイープ GC である。

図4、図7より読み取れるように、**N-body**,**deltal**
では、閾値 C_{th} を適切に選べば、総実行時間で Boehm GC より良いパフォーマンスが得られる。それぞれ、最大で 2.3 %, 18.9 % の実行時間の短縮が観測されている。これは、図5 や図 8 から分かるように、**N-body**, **deltablue** では、GC 時に完全な空きページとして処理されるページが多く、そのようなページが高速なリニアアロケーションに使われるためである。$C_{th} = 0$ の場合は、マークスイープ方式で空のページをリニアアロケーションに使用するというだけの単純な方法であるが、それでもアプリケーションによっては、従来のマークスイープ + フリーリストからのアロケーションよりも合計コストが小さいことが実験結果よりわかる。

逆に、**cube** では、完全な空きページが少ないため、RCGC でのアロケーションの多くはフリーリストからのアロケーションになっている。そのため、特に $C_{th} = 0$ で、アロケーショ

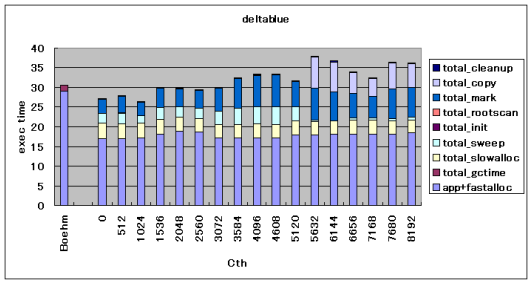

図 4: deltablue: 実行時間

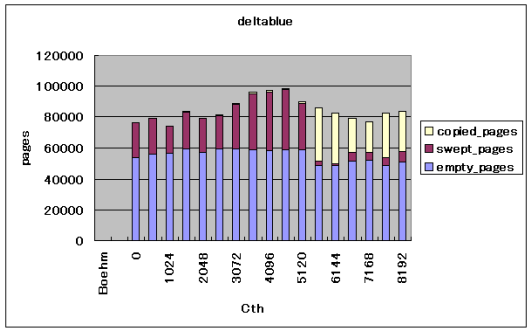

図 5: deltablue: スイープ、コピーされた/空であったページ数

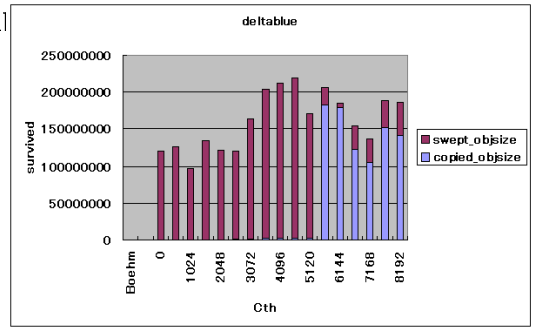

図 6: deltablue: オブジェクトの生存量

C_{th}	Boehm	0	1024	8192
実行時間	30.45	28.07	24.71	32.36
(比)	1.00	0.92	0.81	1.06
GC 時間	1.52	7.27	4.48	12.65
GC 回数	10	11	11	22
空ページ	N/A	70%	81%	63%

表 2: 実験結果: deltablue

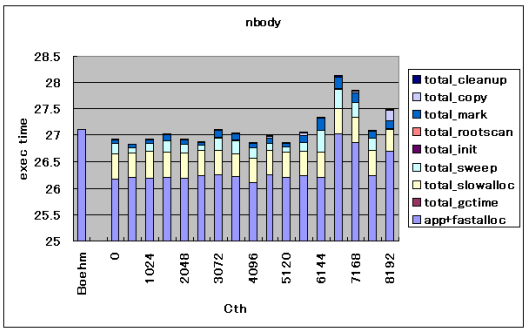

図 7: N-body: 実行時間

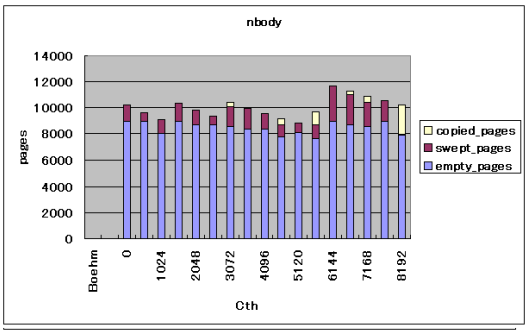

図 8: nbody: スイープ、コピーされた/空であったページ数

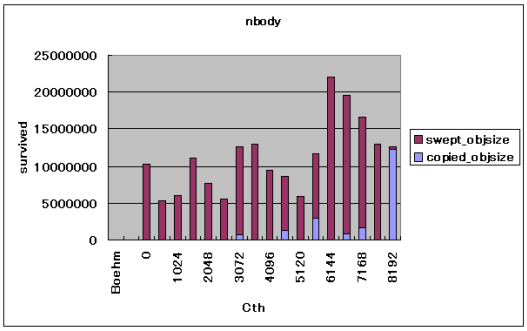

図 9: N-body: オブジェクトの生存量

C_{th}	Boehm	0	1024	8192
実行時間	27.11	26.99	26.90	26.86
(比)	1.00	0.99	0.99	0.99
GC 時間	0.0019	0.29	0.23	0.28
GC 回数	1	5	5	10
空ページ	N/A	87%	88%	88%

表 3: 実験結果: N-body

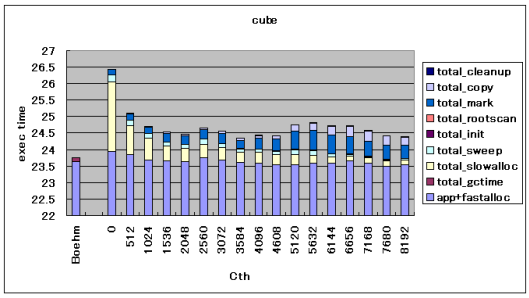

図 10: cube: 実行時間

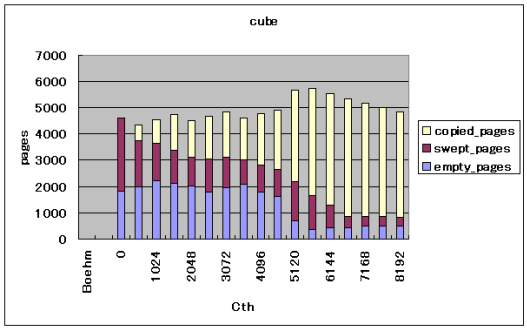

図 11: cube: スイープ、コピーされた/空であったページ数

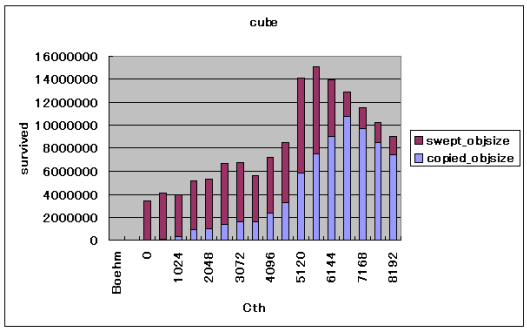

図 12: cube: オブジェクトの生存量

C_{th}	Boehm	0	1024	8192
実行時間	23.76	25.36	24.15	24.31
(比)	1.00	1.06	1.02	1.02
GC 時間	0.12	0.38	0.34	0.65
GC 回数	8	9	10	19
空ページ	N/A	40%	50%	10%

表 4: 実験結果: cube

ン速度の遅さが、パフォーマンスに大きく影響している。

GC 実行時間については、RCGC の方が数倍遅いが、これは主に以下の三つの要因が考えられる。まず、Boehm GC では、スイープを GC 時ではなく、アロケーション時に行うことである (Lazy Sweeping)。RCGC ではスイープを GC 時に行うため、その分 GC に時間がかかる。二つ目は、RCGC ではいろいろなサイズのオブジェクトが混在しているため、オブジェクトの先頭以外の部分を指すポインタからオブジェクトの先頭のアドレスを求めるコストが大きいことである。Boehm GC では、一つのページには同じサイズのオブジェクトしか存在しないため、高速にオブジェクトの先頭アドレスを求めることができる。三つ目は、オブジェクトをマークする際、ER set にオブジェクトを指すポインタの位置を登録する際のオーバーヘッドである。

マークスイープとコピーのトレードオフを解消する最適な閾値 C_{th} については、実験結果のグラフでの実行時間が一番小さいところが最適であるとは言えない。なぜならば、GC 時の生存オブジェクト量が GC 時間に、従ってパフォーマンスに大きく影響しているからである (生存オブジェクト量のグラフと実行時間のグラフを比較すれば明らかである)。生存オブジェクト量が異なるのは、3.2 節で示したように、GC の起動タイミングが閾値に依存して異なるためである。

次章で、最適な閾値についての考察・実験をする。

5 望ましい閾値の設定

4 節では、閾値を様々に変化させた場合の性能評価を行なった。GC 時に空になるページが多い場合、RCGC は従来の保守的マークスイープ GC よりもパフォーマンスがよいことが分かった。

この節では、空でないページについて、コピーかスイープかの選択を適切に行う方法を考察する。そのために、動的メモリ管理コストのモデル化を行い、コピーとスイープのコストを見積もる方法を考える。

GC のコストは、以下の単位コストとそれぞれの操作の対象となるオブジェクトの量より見積もることができる。

1. 単位オブジェクトサイズ当りの、スイープにかかるコスト
2. 単位オブジェクトサイズ当りの、コピーにかかるコスト
3. フリーリストからのアロケーションのコスト
4. リニアアロケーションのコスト

簡単のためキャッシュや TLB の影響を無視するとして、以上のうち、1, 3, 4 は、オブジェクトの個数にのみ依存すると考えられ、2 はオブジェクトの個数と合計サイズの両方に依存すると考えられる。そこで、ページ当りのコストは、オブジェクトの合計サイズと個数のみに依存すると近似する。つまり、コピーとスイープのどちらが有利であるかを判断するために、実験結果をもとに、各ページのコピーおよびスイープのコストを

$$cost = f(L, N)$$

L は生存オブジェクト量

N は生存オブジェクト数

の形であらかじめ求めておくことを考える。

f を求めるため、固定オブジェクトサイズ、固定ヒープサイズで、生存オブジェクト量が同じページをひたすら作成するベンチマークプログラムを用い、様々な生存オブジェクト量に対する (1) マークのコスト、(2) スイープのコスト、(3) コピーのコストを測定した。

横軸がページサイズに対する、生存オブジェクト量の割合であり、縦軸が 1 ページ当りの各操作の実行時間を表す。実験は IBM Netfinity 7100 上で行われた。CPU は Pentium III Xeon 700 MHz, OS は Linux 2.4.18 である。ページサイズは 8192 バイトである。オブジェクトの

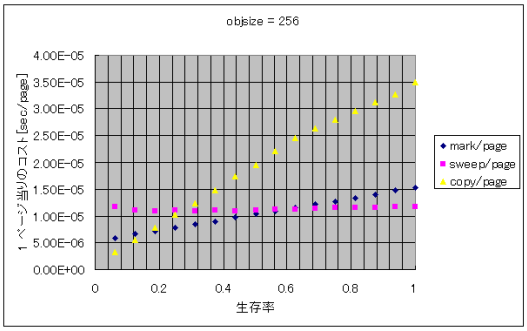

図 13: 生存オブジェクト量に対する、各操作のコスト (objsize = 256)

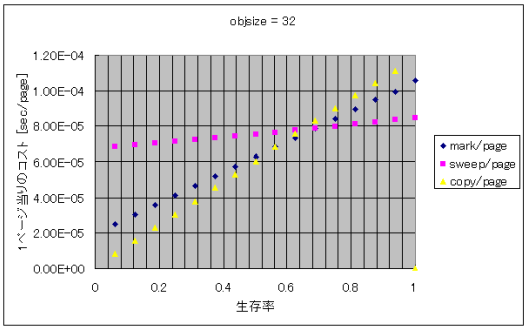

図 14: 生存オブジェクト量に対する、各操作のコスト (objsize = 32)

サイズが 256 バイトと 32 バイトの場合の測定結果を、それぞれ 図 13, 図 14 に示す。

単純な方法としては、コピーとスイープのコストの交点を閾値 C_{th} にとることで、RCGC アルゴリズムのもとでの GC 時間を最小にすることができると考えられる。しかし、実験結果によると、コピーとスイープのコストの交点はオブジェクトのサイズによって異なるため、単一の閾値 C_{th} による方法では不十分である。

図 13, 図 14 を含む、様々なオブジェクトサイズに対するデータをもとに、最小二乗法により、コストのグラフの傾きと y 切片を算出した。その結果、傾きは図 15, y 切片は 図 16 のようになった。

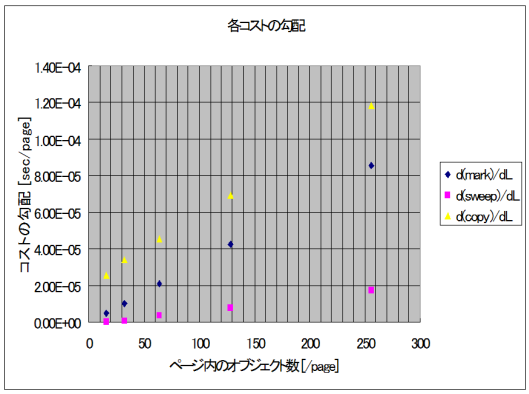

図 15: 各コストの、オブジェクト生存率に対する勾配

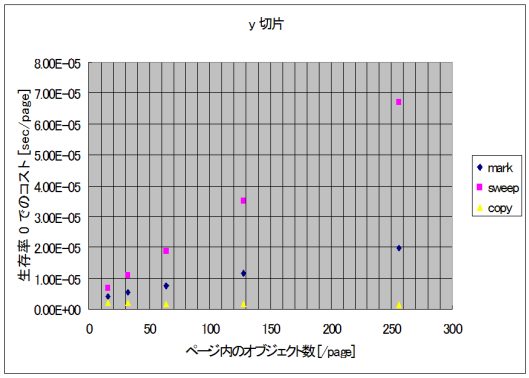

図 16: オブジェクト生存率がゼロの時のページ当りのコスト

実験結果より、IBM Netfinity 7100 上で、コ

ピーおよびスイープのコストは、それぞれ

$$\begin{aligned} C_{copy}(L,N) &= (0.38*N+21)*L \\ &\quad +(-0.0030*N+2.0)[\mu sec] \\ C_{sweep}(L,N) &= (0.073*N-1.4)*L \\ &\quad +(0.25*N+2.7)[\mu sec] \end{aligned}$$

で表せることが分かった。

今後、この結果に基づいた戦略選択方式を実装・評価する予定である。また、ここまでの実験では、アロケーションのコストが含まれていないことに注意されたい。今後、アロケーションも含めたコストモデルを作成し、評価を行う予定である。

6 関連研究

Boehm らの 保守的マークスイープGC[4] は、メモリ上のワードを全て曖昧なポインタとして扱う GC である。C や C++ でも使えることや、プログラミングインターフェースが最も簡単で使いやすいのが特徴である。ただし、マークスイープ方式であり、フリーリストよりアロケーションが行われるため、アロケーション速度に難がある。

また、曖昧なポインタの存在下でコピー GC を行うものとして、Bartlett の Mostly-copying collector [2] がある。これは、曖昧なポインタに指されているオブジェクトは固定、それ以外のオブジェクトをコピー GC で回収する方式である。コピー方式であるため、連続な空き領域が得られ、リニアアロケーションが可能である。ただし、ヒープは半分しか使えず、また、生存オブジェクト量が多い場合、GC が遅いという問題点がある。

また、オブジェクトの少ないページに関してスイープの時間を短縮する研究も存在し [5]、LaTTe Java VM に搭載されている [6]。これは、生きているオブジェクトが少ないページについて、生きているオブジェクトのアドレスを覚えておき、アドレス順にソートすることで、生きているオブジェクトに挟まれた空き領域を、一息で Sweep できる、というものである。ソートには $O(n \log n)$ の時間がかかるが、n が非常に小さい場合のみ行うので、それほどコストはかからない. RCGC は、生存オブジェクトが少ないページに関してはコピーをする、という点がこの研究と異なる。

また、ヒープの一部にコピーを適用するという研究がある [3]。この研究の狙いは、重い Full Compaction を避け、一部のみ Compaction を行うことにある。あらかじめ、ある領域を、コピーする領域、として定め、マークコピー, それ以外の領域については、マークスイープを適用する。コピーする領域については ER set を使い、オブジェクトを指すポインタを記録しておく点については、本研究と同じである。RCGC は、コピーを適用する領域を生存オブジェクト量に応じて決定するという点で、この研究とは異なる。

また、GC のコストを削減する一般的な手法の一つに世代別 GC[7] がある。過去に GC を何度も生き抜いたオブジェクトは新たに GC が起こったときも生きている可能性が高いという経験則がある。そこで、古いオブジェクトを生きているものと決め打ちしてしまう。これにより GC 時に探索すべきオブジェクト量を削減することができる。本研究の RCGC アルゴリズムは、与えられた生存オブジェクトに対し効率的に GC を行うものであり、世代別 GC と直交している。両者を組み合わせることにより、更にコスト削減が可能と考えられる。

7 まとめと今後の課題

本論文は動的メモリ管理コストを削減する RCGC 方式の提案を行なった。GC コストとアロケーションコストの合計を削減するために、マークスイープ方式とコピー方式を混合する。生存オブジェクトが多いページについてはアロケーションコストよりも GC コストが支配的であるため、生存オブジェクトを動かさずにスイープを行う。生存オブジェクトが少ないページについては、GC コストよりも将来のアロケーションコストが支配的であるため、コピーを行

うことにより大きな空き領域を作る。これにより高い頻度で高速なリニアアロケーションが可能になる。

コストを小さくするために、各ページについてコピーかスイープを選択する戦略をとる。はじめに、各ページの生存率と定数の閾値を比較するという単純な戦略について述べた。この閾値 $\beta = \frac{C_{th}}{PAGESIZE}$ を 0 としたときはマークスイープ GC と同等であり、1 に近づけるにつれマークコピー GC に近づく。閾値 β が 0 の場合でも、空きページはリニアアロケーションに使われるため、空きとなるページが多いプログラムでは、RCGC がよいパフォーマンスを示すことが実験により示された。

代表的な保守的マークスイープ GC である Boehm GC と比較しても、空きページが多く発生する二つのプログラムでは総実行時間がより小さいことが示された。

この手法では、望ましい閾値をあらかじめ決定しておく必要がある。望ましい閾値を調べるために、様々な閾値についてベンチマークプログラムの性能を計測した。しかし、この実験により最適な閾値が見つかったわけではない。これは閾値が違うと GC のタイミングが異なるため、生存オブジェクト量も変わってしまうためである。

また、より単純な (生存オブジェクト量が毎回の GC で一定であるような) マイクロベンチマークを用いて実験を行った。これにより、コピーおよびスイープのコストを、それぞれオブジェクト生存量とオブジェクト数の単純な関数として表すことができた。この式をもとに、コピーとスイープのコストを見積もり、小さい方を選択することで、RCGC アルゴリズムの元での GC のコストを最小にできると考えられる。ただし、この方法はアロケーションのコストを考慮に入れていないため、合計コストを最小にできるわけではない。今後、アロケーションも考慮にいれたコストモデルを作成し、合計コストを最小にする戦略を考案・評価する予定である。

また、本稿では述べていないが、断片化の具合などもコピーの有利性やアロケーションの速度に影響すると思われるため、それも考慮に入れた戦略を今後考案する予定である。また、参照の局所性による TLB やキャッシュへの影響もパフォーマンスに大きく影響すると考えられるため、それらも考慮に入れた方式を、今後考案する予定である。

参考文献

[1] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[2] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, 1988.

[3] Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Kean Kuiper, and Victor Leikehman. An algorithm for parallel incremental compaction. In *Proceedings of the third international symposium on Memory management*, pages 100–105. ACM Press, 2002.

[4] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[5] Yoo C. Chung, Soo-Mook Moon, Kemal Ebcio?lu, and Dan Sahlin. Reducing sweep time for a nearly empty heap. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–389. ACM Press, 2000.

[6] Latte : An open-source java virtual machine and just-in-time compiler.

[7] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.