プログラミング—言語・基礎・実践— 3－20
（1991. 7. 17）

可変長セル用並列・実時間ごみ集め

鈴木 貢, 寺島 元章
電気通信大学[1] 情報工学科

概要

各種の自動記憶管理機構の中でもごみ集めは、動的なデータ構造を扱うプログラミング言語処理系の具現上必須の機能である。可変長セルを含むデータ構造の多様化や、実時間応答する処理系への要求はごみ集めを質的に変えてきた。本論では可変長セルの並列・実時間ごみ集めを滑べり圧縮法によって行なうための基本的な記憶管理法を提案し、その有効性と実装について論じる。

A Garbage Collection Scheme for Variable Sized Cells in Parallel or Realtime

Mitsugu Suzuki and Motoaki Terashima
Department of Computer Science, The University of Electro-Communications [2]

Abstract

Among automatic storage management mechanisms, garbage collectors (GC in short) are indispensable for impremementations of programming languages supporting dynamic data structures. Variety of date structures, including variable sized cells, and realtime features have changed GCs. In this paper, we present a basic storage management scheme used for parallel or realtime garbage collectors of variable sized cells based on sliding compaction technique, with it's efficiency and actual implementations.

1 初めに

プログラマを記憶管理という問題から解放する事は、プログラムの生産性の向上や、取りにくい虫の発生を抑えるという点で重要である。Lisp や Smalltalk 等の言語の処理系には、ごみ集め (garbage collection 以下 GC と呼ぶ) と呼ばれる自動記憶管理機構が備わっており、プログラマはセル (オブジェクト) の生成死滅を意識する事無くプログラミングする事が出来る。

一方、そのような言語が取り扱うデータタイプが応用分野の広がりとともに多種多様になり、可変長セル (大きさが一定でないセル) を取り扱う事が必須となってきた。可変長セルを取り扱うシステムでは、ヒープの断裂 (fragmentation) を回避するためにヒープの使用領域を圧縮する必要がある。また、仮想記憶上で稼働するシステムの場合には、ワーキングセットを小さくするという観点からもヒープの使用領域の圧縮は重要である。さらに、それらの言語をロボットの制御の様に実時間応答が求められる分野で使用する場合、GC のために計算の途中で実行が、一時的にせよ停止してしまう事は大きな問題である。そこで、GC を計算の実行と並列、或いは時分割に行う事が必要である。(尚、ごみ集めをまとめて行なう古典的な GC を incremental と対比して一括方式と呼ぶ。)

本論では、滑り圧縮を用いて incremantal に可変長セルの GC を行うための方式を示し、その効率に関する考察と、より実用的な応用のための変更について述べる。

2 関連研究

ここで本論に入る前に、今までになされた実時間 GC の研究の主要なものについて概観する。

2.1 並列 GC

Steele[SJ75] は、固定長セルの GC を並列に行なう方式について論じた。GC は印付け・圧縮・回収の三つの段階を繰り返す。この方式は、ヒープの使用領域の圧縮を入れ換え法で実現しているので可変長セルを扱うことは出来ず、また 3.1で述べるセルの世代管理には利用出来ない。また、この方式は多くの排他制御を用いており、汎用プロセッサにおける効率の良い実現は不可能である。湯浅 [Yua85] は、この方式を時分割方式に変形したものを提案した。

Dijkstra らの [DLM+78] は、固定長セルのごみ集めを並列に行なう方式にについて論じた。GC は、印付け・回収の二つの段階を繰り返す。この論文の核心は、いかに印付けを並列に行なうかという事であり、走査法と呼ばれるアルゴリズムが提案されている。このアルゴリズムは、(ヒープの大きさ×リストの最大長) に比例する時間を要するため、実用上に問題がある。この方式の変形として、3 色必要であった印を 2 色に減じた [BA84] がある。

Kung ら [KS77] も、固定長セルのごみ集めを並列に行なう方式にについて論じた。[DLM+78] との相違点は、印付けのアルゴリズムにある。この方式は、古典的なリストの再帰的な辿りをスタックではなく双頭の待ち行列を使って行なっている。また、到達出来ないセルと、到達可能かつ待ち行列に入れられたセルと、到達可能かつ印付けを完了したセルと、自由リストのセルを区別するために 4 色の印を用いている。従って、印付けのためにかかる処理時間は、生きているセルの数に比例する。

日比野 [Hib78] も、固定長セルのごみ集めを並列に行なう方式について論じた。[DLM+78], [KS77] との相違点は、印付けのアルゴリズムである。スタックを使ったリストの再帰的な辿りを計算プロセスからの操作要求が無くなるまで繰り返す。

以上、[DLM+78], [KS77],[Hib78] の 3 つのアルゴリズムでは、ヒープの使用領域の圧縮は行なっていない。

2.2 実時間 GC

Baker[BJ78] は、可変長セルのごみ集めを時分割に行なう方式にを提案した。このアルゴリズムは、基本的には仮想記憶を用いる一括複写圧縮方式の変形である。異なるのは、以下の点である。

- 二次記憶と主記憶の間の複写ではなく主記憶上の等しいサイズの二つの領域間の複写である。
- セルの確保が行なわれる度に scavenging(生きているセルを複写する事) を目標個数行なう。
- 未回収のセルをアクセスした時にそれを scavenging する。
- 新しいセルの確保を複写先の領域から行なう。

この方式の問題点は、セルの生存期間に関係なく全ての生きているセルが複写の対象となる事である。

Liberman ら [LH83] も同様に可変長セルのごみ集めを時分割に行なう方式について論じた。[BJ78] と異なるのは、各セルを世代別に異なる領域に配置し世代に

よってscavengingする頻度を変える点、つまり古いセルほど少ない頻度でscavengingするという点である。この工夫の有効性の根拠は、3.1で述べる。

Ungar[Ung84]は、可変長セルのごみ集めを一括方式で行なう。ただし一括方式であるがscavengingの対象を小さな領域に限ることで、対話環境(Smalltalk-80)でも我慢できる程度の時間(数百ミリ秒)の一時停止でごみ集めが完了すると述べている。ただし、マイクロコードやハードウエアトラップを多用して実現されている。セルを世代別に管理するという点では[LH83]と同じであるが、以下の点で異なる。

- 世代管理を行なうのは比較的新しいセルだけである。
- 古いセルは大きな領域(oldと呼ばれる)に移動してscavengingの対象としない。
- 世代情報は領域ではなくセル自身が持つ。

[LH83]と[Ung84]の二つの方式に共通する問題点は、古い世代のセルからより新しい世代のセルを参照する時、その参照を記録するremembered setというデータ構造が必要であるため、多くの場合無駄な領域が生じる。

3 議論

ここでは、滑べり圧縮法に関する利点や問題点について述べる。

3.1 セルの世代管理

ある程度古いセルは、多くの場合プログラムが終了するまで生き続けるという統計的な事実があり、また、そのようなデータがヒープに占める割合は決して小さくない。それらを回収や圧縮の対象から外す事は、ごみ集めのためにかかる負荷を減らす事に大いに寄与する。こうしたセルの世代別管理は効率の良いGCを設計する上で重要である。[LH83]や、[Ung84]の論文は、この点の指摘にあった。そして、滑べり圧縮には特別な世代管理を行なわなくても自然な世代管理が行なわれるという大きな利点がある。その理由は、滑べり圧縮の場合 図1 の様にセルが古い順にヒープの片方に溜っていくからである。

3.2 滑べり圧縮法とincremental GC

可変長セルを滑べり法によってincrementalに圧縮する方針は、これまでに提案されていない。incremental GCは、従来から複写方式に依っていた。その理由は以下の通りである。

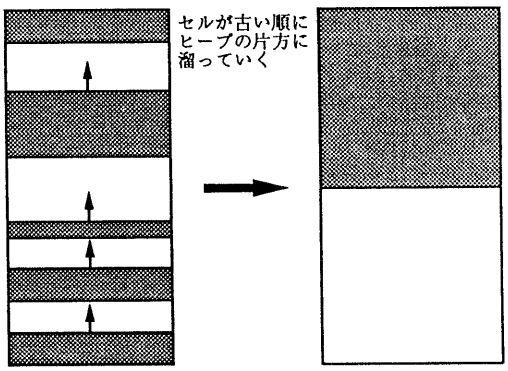

図1: 滑べり圧縮の利点

1. ヒープの一端から生きているセルを移動・圧縮する必要がある。(任意の順では移動出来ない)
2. 移動されたセルを指しているポインタを補正する作業は圧縮の後でなければ行なう事が出来ない。

特に2項は、滑べり圧縮をincremantal GCに応用する上で、大きな問題となる。

3.3 ポインタ補正の解消

ポインタ補正が必要であるのは、ポインタが指しているセルが移動してしまうからである。そこで、セルを移動する部分と移動しない部分に分離する事を考える。具体的には、セルを参照は間接ポインタを介して行なう。そして移動に伴うセルの位置の変化は、間接ポインタを補正する事で吸収する。以下, この間接ポインタのためのデータ構造をディスクリプタ、また、ヒープに配置されたデータをセルの中身と呼ぶ事にする。

3.4 ディスクリプタのソート

ヒープの一端からセルの移動を行なう方法について述べる。ごみ集めで一時停止している期間を小さくするためには、セルの中身の移動と、それに伴うディスクリプタのポインタ補正を出来るだけ短時間に行なう必要がある。第1案は、セルの移動する部分から移動しない部分(つまりディスクリプタ)へのポインタを用意し、ヒープを上から走査し、対応するディスクリプタのポインタ補正をする事である。しかし、この方式は6.2で述べる共有セルを考えるとヒープの構造が

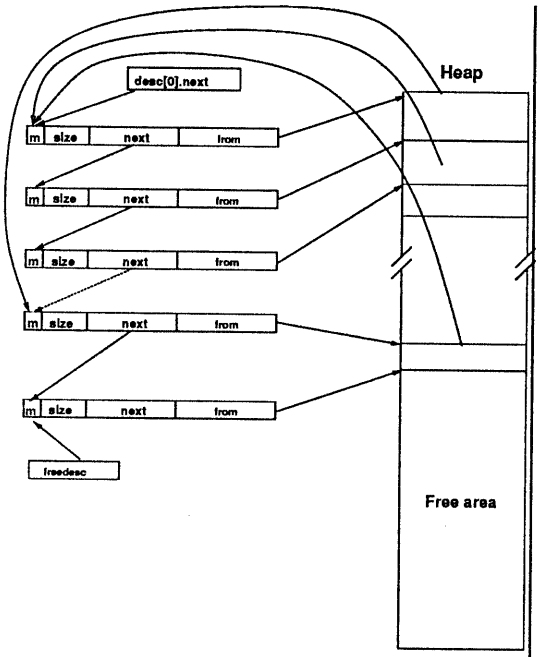

図 2: ディスクリプタを含めた参照関係

均一でない事から不適切である。。第 2 案は、ディスクリプタを GC の度にソートする事である。ディスクリプタのソートにかかる時間が、生きているセルの数を n として、n × log(n) に比例する点と、ソートしたディスクリプタのためのデータ構造が一時的に必要であるので得策とは言えない。そこで、ディスクリプタに別のディスクリプタを指すフィールドを設け、そのポインタによってディスクリプタの連鎖を構成し、ディスクリプタが指しているセルのアドレスの順に並ぶ様にする。新しくセルを確保する時は、この連鎖の最後にそのセルのディスクリプタを接続する。以下ディスクリプタのポインタを補正し、next フィールドを参照して次のディスクリプタに移る事をディスクリプタの連鎖を手繰ると言う事にする。このデータ構造によってディスクリプタのソートのための時間は不要となる。また、ディスクリプタを手繰りながらセルの移動・回収を行なうので、並列型の場合は計算プロセスの回収中のセルへのアクセス不能な時間、時分割型の場合は GC のための一時停止時間は、セルの中身を移動している時間とディスクリプタを手繰る時間の合計である。
この参照関係を図示すると、 図 2 の様になる。

4 アルゴリズム

ここで、GC のアルゴリズムを示す。記述には Pascal を用いる。ただし、印付けのアルゴリズムについては、2.1で挙げたもののうちのいずれかを用いるものとする。また、ディスクリプタの確保や回収の操作の詳細は述べない。これらは一般的なリスト操作で出来る。(ただし、並列 GC の場合ならば排他制御が必要である) セルの中身の移動に関しても同様である。

4.1 データ構造

ディスクリプタのデータ構造と、ヒープのデータ構造をそれぞれ 図 3 と 図 4 に示す。セルの生存印やサイズ等の情報は、ディスクリプタの中に置かれているものとする。ここで、以下の主張が成り立つものとする。

1. ディスクリプタへのインデクスが 0 である時は、nil(何も指さない) とする。

2. ヒープの一番上に配置されているセルのディスクリプタは desc[0].next で指されるものとする。

3. freedesc は、ディスクリプタの連鎖の最後尾を指す。

4. ヒープの自由領域は、desc[freedesc] で指されていて、一つのセルの様にサイズや開始位置の情報を持つものとする。

5. 新しいセルのディスクリプタは、ディスクリプタの連鎖の最後尾から 2 番目に割り当てられる。(最後尾のディスクリプタは自由領域の情報を持つ。)

4.2 セルの確保

セルを確保するためのアルゴリズムを 図 5 に示す。これは、自由領域を必要なサイズの分まで縮小し、新しく確保したディスクリプタに残りを割り当てる。エラーチェックは省いてある。

4.3 セルの回収

セルを回収するためのアルゴリズムを 図 6 に示す。4 行目から 7 行目のループを GC のサイクルと呼ぶ事にする。

```
type
    { ディスクリプタの配列へのインデクス }
    descindex = 0..descmax;

    { ディスクリプタの構成 }
    desctype = record

        { 印 }
        m : marktype;

        { セルのサイズ }
        size : integer;

        { セルの先頭位置 }
        from : heapindex;

        { 次のセルのディスクリプタ }
        next : descindex;
    end;

var
    { ディスクリプタ }
    desc : array[descindex] of desctype;

    freedesc : descindex; { 本文中を参照 }
```

図 3: ディスクリプタのデータ構造

```
type
    { ヒープの配列へのインデクス }
    heapindex = 0..heapmax;

    { タグの種類 }
    tagtype = (ptr, ...);

    { ヒープの要素 }
    heaptype = record
        case tagtype of
        ptr : (p : descindex); { セルを指す時 }
        .....
    end;

var
    { ヒープ }
    heap : array[heapindex] of heaptype;
```

図 4: ヒープのデータ構造

```
function alloc(reqsize : integer) : descindex;
{  大きさが reqsize のセルを確保する  }
var
    newfree : descindex;
begin
    newfree :=
        ディスクリプタをフリーリストから確保;

    with desc[newfree] do begin
        size := desc[freedesc].size - reqsize;
        from := desc[freedesc].from + reqsize;
        next := nil;
    end;

    with desc[freedesc] do begin
        next := newfree;
        size := reqsize;
        m := 生存;
    end;

    { 確保されたセルのディスクリプタ }
    alloc := freedesc;
end;
```

図 5: セルの確保

```
procedure gc;
{  GC の本体  }
begin
    while TRUE do begin
        印付け;
        recraim;
    end;
end;

procedure recraim;
var
    dp, tmp : descindex;
    hp : heapindex;
begin
    dp := 0;
    hp := 0;
    while next ≠ freedesc do
        with desc[dp] do begin
            if desc[next].m = 生存 then begin
                desc[next] で指されたセルを hp に移動;
                hp := hp + desc[next].size;
                dp := desc[next].next;
            end else begin
                tmp := next;
                next := desc[next].next;
                desc[tmp] を回収;
            end;
        end;
    { この時点で自由領域のディスクリプタに達している }

    with desc[freedesc] do begin
        from := hp;
        size := heapmax - hp + 1;
    end;
end;
```

図 6: セルの回収

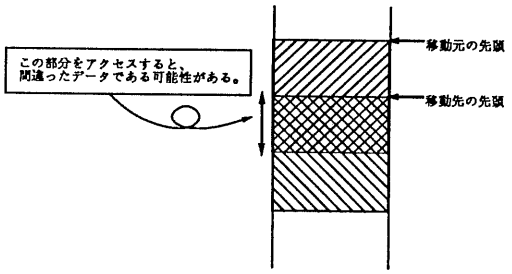

図 7: 移動によってセルが重なり合う場合

4.4 並列処理について

並列処理のためには、リストプロセスと GC プロセスが、各々のセルやディスクリプタのフリーリストを巡って排他制御を行うようにしなければならない。個々のセルに排他制御が必要であるのは、以下の理由による。

- 同じセルに GC による移動と計算プロセス側の書き換えが同時に起きた時に、正しく書換えが行なわれる事を保証する。
- 移動によってセルが重なり合う場合、間違ったデータをアクセスする可能性がある。(図 7 を参照)

4.5 時分割処理について

[BJ78] の様に時分割処理を実現するためには、計算と GC の二つのコルーチンを用意し、両者を以下に述べるタイミングで両者を切り替える。

- alloc が呼ばれる度に GC のコンテクストに切り替える
- GC のコンテクストでは、印付けやセルの回収が決められた量 (活動目標) だけ完了したら、計算のコンテクストに切り替える

5 静的な評価

この方式の静的な性能評価を行なう。これまでは、具体的な印付けのアルゴリズムを示さなかったが、ここでは [KS77] で示されている印付けアルゴリズムを用いると仮定する。また、3.1で述べた世代別管理による効果は考えない。

5.1 パラメタの定義

議論を簡単にするため、一つのセルのサイズの平均を$\overline{s}$と仮定する。そして、ヒープの容量と、ディスクリプタの容量の比ははは$\overline{s}$: 1 であるとする。($heapmax$: $descmax = \overline{s}$: 1) そして、生きているセルの最大のものの容量を s_{max}とする。GC の終了時に生きているセルの個数を n_{live}、配置されているセルの個数を n_{alloc}とする。ヒープの一つの要素 (図 4 で定めた record の一つ) を移動するのにかかる時間を t_v、ディスクリプタを一つ手繰るのにかかる時間を t_dとする。図 6 を使って説明すれば、

$$t_v = \frac{\text{20 行目を実行する時間}}{\text{セルのサイズ}}$$

$$t_d = \text{19 〜 27 行目の実行時間} - \text{20 行目の実行時間}$$

となる。一つのセルを印付けするのにかかる時間を t_mとする。

5.2 ごみ集めにかかる手間

ごみ集めにかかる時間Tは、印付けにかかる時間をT_m、回収・圧縮にかかる時間をT_rとすれば、

$$T = T_m + T_r$$

である。T_mは、[KS77] の場合、生きているセルの個数に比例する時間である。すると、

$$T_m = t_m \times n_{live}$$

となる。T_rは、ディスクリプタの連鎖の全てを手繰る時間と、生きているセルの中身を全て移動する時間の合計である。つまり、

$$T_r = t_d \times n_{alloc} + t_v \times \overline{s} \times n_{live}$$

である。よって、

$$T = (t_m + t_v \times \overline{s}) \times n_{live} + t_d \times n_{alloc}$$

実際には、$\overline{s}$がある程度大きければ、

$$t_m + t_v \times \overline{s} \gg t_d$$

となり、

$$T \approx (t_m + t_d + t_v \times \overline{s}) \times n_{live} \qquad (1)$$

つまり、ごみ集めにかかる時間は生きているセルの個数に比例すると言える。

5.3 計算の一時停止時間

時分割GC の場合、各 GC のサイクルの終了時に次のサイクルの終了時までに配置可能であるセルの個数は $descmax - n_{live}$であるので、alloc が呼ばれる度に

$$\overline{tp} = \frac{T}{descmax - n_{live}} \qquad (2)$$

に相当する時間の分だけ印付けや回収の作業を行なうと、GC による一時停止時間は平均化され一括型の GC に入る事がない。その時の印付けや回収の個数は、t_m, t_d, t_v の比と、$\overline{n}$ の値で決定する事が出来る。$\overline{tp}$ は、一回 alloc が呼ばれる度に GC のために一時停止する時間の平均値である。言い替えれば、4.5で述べた活動目標である。(1) から、T は n_{live} に比例するので、$\overline{tp}$ も n_{live} に比例する。つまり、平均一時停止時間は生きているセルの個数に比例する。
また並列 GC の場合、計算プロセスが平均時間 $\overline{t_a}$ 毎にセルを確保するものとすれば、

$$\alpha = \frac{\overline{tp}}{\overline{t_a}}$$

は GC プロセスと計算プロセスの最適活動量比となる。α を元にして、GC プロセスと計算プロセスのタイムスライスの比を決定したり、共有メモリの占有率を設定する。
並列 GC の場合、GC のために計算が一時停止する時間の最大値 tp_{max} は、

$$tp_{max} = t_d + t_v \times s_{max}$$

である。この状態は、最大容量のセルへのアクセスと GC の回収が重なった時に起きる。
時分割 GC の場合は、$\overline{tp}$ と tp_{max} のうち大きい方が一時停止時間の最大値である。

5.4 平均一時停止時間とセル領域の大きさの関係

(1) によると、T は $descmax$ のかかる項を持たないので、(2) から、$\overline{tp}$ は $descmax$ が大きいほど小さくなる。よって時分割 GC の場合は、計算の GC による平均一時停止時間は、ディスクリプタとヒープが大きいほど短くなる。並列型 GC の場合は、それらが大きいほど GC プロセスの最適活動比αはは小さくなる。

6 拡張

これまでに述べたアルゴリズムやデータ構造等に変更を加えてより効率的、実用的なものにする事について議論する。

6.1 世代別管理

ある程度古いセルの回収を放棄する GC の戦略がある。印付けの範囲を新しい世代のセルに限る方式については、使用する印付けのアルゴリズムに依存するので、ここでは述べない。
回収の手間は、以下の二つから成る。

1. ディスクリプタを手繰る手間
2. セルの中身を移動する手間

このうち後者は、回収を放棄したセルは不動セルである可能性が大きいので特別な方式を導入しなくても消去される。不動セルに関しては、寺島の [TS89] で補正値が 0 である塊として説明されている。前者の手間は、次の様な方式で省く事ができる。

1. 何らかの方針に従い「これ以上古いセルは回収の対象としない」というセルを決める。
2. 回収の開始点を desc[0].next でなく上記で決めたセルにする。

6.2 共有セル

文字列処理や行列演算では、大きなセルの一部を一つのセルの様に扱う事がある。このようなセルを(部分) 共有セルと呼ぶ。図4 で示したデータ構造は、均一なレコードの配列なので、このような機能を組み込む事が可能である。確保した共有セルを元になったセルを親と呼ぶ事にする。共有セルは以下の様なものである。

1. 共有セルは、その親の部分集合である。
   (二つのセルにまたがらない)
2. 共有セルを親とする共有セルも許される。

ある共有セルの親を遡っていく時、親が存在しないセルに遭遇するが、これを先祖と言う事にする。そして、先祖とそこから直接あるいは間接に作られた全ての共有セルの集合を家族と呼ぶ事にする。また、共有セルの家族のうちで、生存しているセルを群と呼ぶ事にする。共有セルを許すデータ構造を扱うためには、以下の様にする。

1. 共有セルを確保する手続き calloc を別に用意する。(この手続きでは、ディスクリプタを確保するだけでヒープは確保しない)
2. calloc で共有セルのディスクリプタをディスクリプタの鎖に加えるときに、ソートされた順番を守るようにする。
3. 回収時に共有セルの群をまとめて移動する。(並列処理の場合は、一時的に群の全てのセルに排他制御の lock をかける)

3 項は、書換えが正しく行われるために必要である。

6.3 CONS セル

このGCをLisp処理系に応用する場合、ディスクリプタとセルの中身という組み合わせでCONSセルを実装すると、使用領域量及びアクセス時間の点で不利である。そこで、次の対策が考えられる。

1. CONSセルを別の領域にとる
2. CONSセルとディスクリプタを同じ構造で表現し、タグで振り分ける。(共用体とする)
3. CDRコーディングを用いる

3項は、リストの途中を指すときに共有セルを確保しなければならないので、あまり得策ではない。また、1項は、ディスクリプタとCONSセルの領域の割合を柔軟に決定出来ない点で問題である。

6.4 sizeフィールドの省略

セルは連続して配置されているので、次に配置されているセルの中身の開始点(desc[desc[i].next].from)との差からそのセルのサイズを知る事が出来る。ただし、共有セルを許す場合は、次の制約を設ける。

1. 共有セルの先祖が最後まで生き残っている必要がある。
2. 先祖だけ特別な印をつけられている必要がある。

前者は回収時の工夫で、後者はディスクリプタにフィールドを付加する事で解決できる。

7 結論

incrementalなごみ集めを、いままで採用されなかった滑り圧縮を用いて行うための方式を提案した。滑り圧縮では、世代別管理を行なうに際して、そのためのデータ構造や操作が不要である点が他の圧縮方式に比べて優れている。ごみ集めにかかる時間は、印付けのための手間を除けば生きているセルの数に比例する。また、ポインタ補正の段階を持たないので、ごみ集めのために一時停止する時間は最大容量のセルの移動にかかる時間である。また、時分割型のアルゴリズムにも変更可能であるので、計算とGCの両プロセス間の排他制御に伴うオーバヘッドを無くす事が出来るため、単一プロセッサ上でも効率よく実装出来る。

参考文献

[BA84] Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Tran. Programming Languages and Systems*, Vol. 6, No. 3, pp. 332–344, Jun 1984.

[BJ78] Henry G. Baker Jr. List processing in real time on a serial computer. *Comm. ACM*, Vol. 21, No. 4, pp. 280–294, April 1978.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *CACM*, Vol. 21, No. 11, pp. 966–975, November 1978.

[KS77] H. T. Kung and S. W. Song. An efficient garbage collection system and its correctness proof. *Proc. 18th Annual Symposium on Foundation of Computer Science*, pp. 120–131, 1977.

[LH83] Henry Liberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Comm. ACM*, Vol. 26, No. 6, pp. 419–429, June 1983.

[SJ75] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *CACM*, Vol. 18, No. 9, pp. 495–508, September 1975.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance strage reclamation algorithm. *SIGSOFT/SIGPLAN*, pp. 1157–167, April 1984.

[Yua85] Taiichi Yuasa. Realtime garbage collection on general-purpose machines. 日本ソフトウエア科学会第 2 回大会論文集, pp. 181–184, 1985.

[TS89] 寺島 元章, 佐藤 和美. 可変容量セルの効果的なくず集めに付いて. 情報処理学会論文紙, Vol. 30, No. 9, pp. 1189–1199, Septebmer 1989.

[Hib78] 日比野 靖. 並列ガーベジコレクションアルゴリズムと lisp への適用. 信学技報, No. 32, pp. 21–30, 1978.