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, 2000 年 3 月

少量のスタックで大部分を深さ優先順にコピーするゴミ集め方式*

八杉 昌宏† 伊藤 智一‡ 小宮 常康§ 湯淺 太一¶

平成 12 年 3 月 21 日

概 要

本研究では，深さを限定した少量のスタックを用いて大部分のオブジェクトを深さ優先順にコピーするゴミ集め (GC) 方式を提案する．代表的な GC 方式の一つであるコピー方式では，必要な (生きている) オブジェクトのみをヒープの別の部分空間へコピーすることで不要なオブジェクト (ゴミ) が占めるメモリ領域を自動的に再利用する．従来のコピー方式では，オブジェクトへの参照を幅優先順に辿ってコピーを行うため，GC 処理や GC 完了後の計算におけるオブジェクトに対するメモリアクセスの局所性が劣化していた．多くの場合，深さ優先順にコピーすればメモリアクセスの局所性が高められるが，そのために必要なスタックの深さは，最悪の場合，生きているオブジェクトの個数に比例する．スタックのために余分なメモリ領域を必要としない方式も提案されているが，処理が複雑となり処理性能に難点があった．一方，提案する方式は高速な処理を特長とし，128 バイト程度のスタックを追加することでメモリアクセスの局所性を改善する．

1 はじめに

自動的なメモリ領域管理を行うゴミ集め (GC) を用いることで，プログラマはメモリ領域管理の負担から解放されアルゴリズムに関する記述など本質的な作業に専念することができる．また，GC を用いることで明示的なメモリ領域管理を行うよりも効率よいプログラムの実行が可能な場合もある．

代表的な GC 方式の一つであるコピー方式では，必要な (生きている) オブジェクトのみをヒープの別の部分空間へコピーすることで不要なオブジェクト (ゴミ) が占めるメモリ領域を自動的に再利用する．この方式ではオブジェクトが占めるメモリ領域を確保するためのヒープの部分空間を同じサイズで二つ準備し，使っていた部分空間から空の部分空間へオブジェクトをコピーし，コピーされたオブジェクトのみを残すことで GC を行う．二つの部分空間の役割は GC の度に入れ替わることになる．この場合，GC 中を除いてはヒープの半分しか利用されず，生きているオブジェクトのメモリ領域サイズの合計の 2 倍以上のヒープが必要となる．しかし，コピーの結果，連続した空きメモリ領域が得られ，その後生成されるオブジェクトのための領域の割り付けが高速化できる．

従来の代表的なコピー方式では，オブジェクトへの参照を幅優先順に辿ってコピーを行う．これは，コピー先の空間のある範囲を「必要だがまだコピーしていないオブジェクト」や「まだコピー先を指していない参照」の集合を管理するためのキューとして有効利用することで，それ以上のメモリ領域を必要としないためである．しかし，幅優先順にコピーを行うため，GC 処理自体や GC 完了後の計算においてオブジェクトに対するメモリアクセスの局所性が劣化するという問題があった．これは，応用プログラムには，多くの場合，オブジェクトへの参照を深さ優先順に辿る (あるいは生成する) アルゴリズムが用いられるためである．

---

*Mostly-Depth-First Copying Garbage Collection with Small Stack Space

†Masahiro Yasugi 京都大学大学院情報学研究科 yasugi@kuis.kyoto-u.ac.jp

‡Tomokazu Ito 京都大学工学部情報学科

§Tsuneyasu Komiya 京都大学大学院情報学研究科

¶Taiichi Yuasa 京都大学大学院情報学研究科

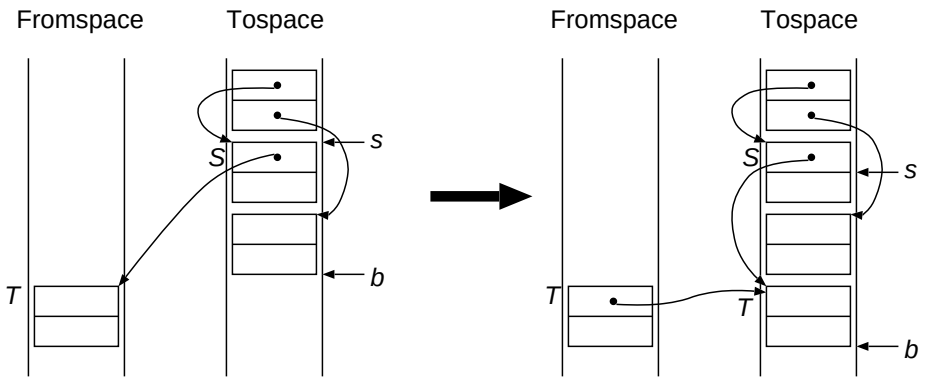

図 1: Breadth-first copying collection.
(S から T への参照)

オブジェクトを深さ優先順にコピーすれば，多くの場合，メモリアクセスの局所性が高められるが，そのために必要となるスタックの深さは，最悪の場合，生きているオブジェクトの個数に比例するという問題があった．このため，スタックのために余分なメモリ領域を必要としない方式 [中島 95] も提案されているが，複雑な処理を必要とするため処理性能に難点があった．一方，提案する方式は単純にスタックを用いた深さ優先順のコピーと同等の高速な処理を特長とし，128 バイト程度のスタックを追加することでメモリアクセスの局所性を改善する．基本的なアイデアは，スタックを用いてできるだけ深さ優先順にコピーを行うが，万一スタックの深さが許される限度を超えるような場合にはスタックを一度空にして深さ優先順のコピーが再開できるように (幅優先順コピー方式と同様に) コピー先の空間のある範囲をキューとして利用するというものである．

2　幅優先順にコピーする従来のゴミ集め方式

幅優先順にコピーする従来の GC 方式 [Che70] は図 1 のような方式である．ヒープの二つの部分空間をそれぞれ Fromspace，Tospace と呼ぶことにする．GC を開始する前は，Tospace にオブジェクトが生成され (つまり Tospace の空きメモリ領域からオブジェクト用のメモリ領域を割り当てる)，Fromspace は空であるとする．Tospace にオブジェクトを生成するための空き領域が足りなくなれば，GC が起動される．

GC の処理では，まず Fromspace と Tospace の役割を交換し，Tospace の空き領域の先頭を指すポインタ (図 1 の b) と，Tospace の未スキャン領域の先頭を指しているポインタ (図 1 の s，詳細は後述) が，Tospace の先頭を指すようにする．

GC の処理では，各ルート (計算に直接利用されている変数で参照を保持しているもの) の Fromspace 内のオブジェクトへの参照を Tospace にコピーしたオブジェクトへの参照に修正する．その際，まだ Tospace のコピーが存在していない場合には，ポインタ b が指す Tospace の空き領域にオブジェクトをコピーし，b を更新するとともに，Fromspace のオブジェクトやそのヘッダの適当な位置にコピー先アドレスを上書きしてコピー先を指示する．これで同じオブジェクトを 2 回コピーしてしまわないようにできる (図 1 の右図のオブジェクト T のようにする .)

すでに Tospace にコピーされたオブジェクト中に存在する「Fromspace 内のオブジェクトへの参照」についても Tospace を向くように修正が必要であるので，修正のためのスキャンが済んでいない領域の先頭を指しているポインタ (図 1 の s) を進めながら，順に Tospace 内のオブジェクトをスキャンする (図 1 では，左図のオブジェクト S の第一要素がスキャンされ，図 1 右図のように Tospace のオブジェクト T を指すように修正される .) つまり，s と b に挟まれた範囲が「まだコピー先を指していない参照」(「必要だがまだコピーしていないオブジェクト」も表している可能性がある) の集合を管理するためのキューとして利用される．この方式の場合，先にコピー

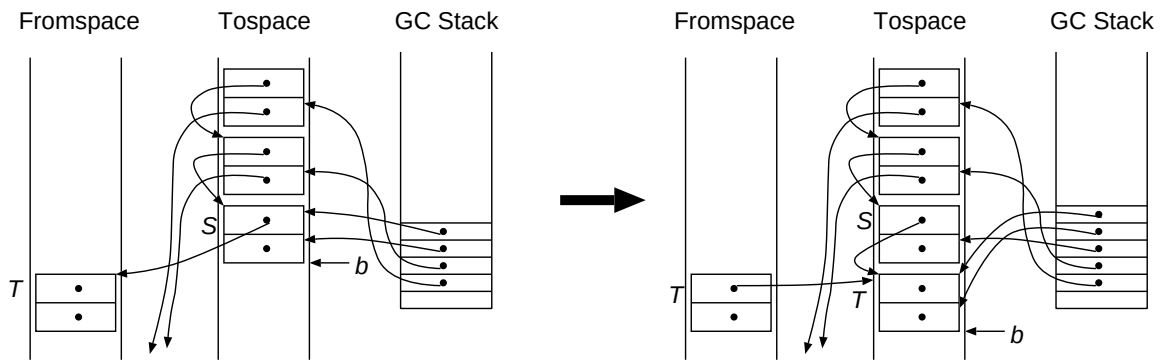

図 2: Depth-first copying collection.

したオブジェクトから先にスキャンされるため，幅優先順にコピーされることになる．キューのためのメモリ領域を別に準備しなくてよいため，未修正の参照の個数とは無関係に s と b の二つのポインタのみ準備してあればよい．一般には，s が指すワードのみを見て参照かどうかは判定できないので，その場合は s をオブジェクト単位で進めながらオブジェクト内の参照すべてについて修正していけばよい．

すべてのルートの処理が完了し，かつ s が b に 追い付いた時点でルートから到達可能なオブジェクト (生きているオブジェクト) はすべて Tospace にコピーされている．この時点で，Fromspace 内のオブジェクトはすべてゴミなので Fromspace は全体が空き領域となり，Tospace も b 以降が空き領域となっている．よって，GC の処理を完了し，通常の計算を再開する．

3　深さ優先順にコピーするゴミ集め方式

3.1　単純にスタックを用いた深さ優先順コピー方式

単純にスタックを用いて深さ優先順にコピーする GC 方式を図 2 に示す．Fromspace，Tospace や，Tospace の空き領域の先頭を指しているポインタ (図 2 の b) については幅優先順の場合と同様である．幅優先順のコピー方式と同様に，ルートから到達可能なオブジェクト (生きているオブジェクト) はすべて Tospace にコピーする．コピーされるオブジェクトについてのみ考えれば，コピーの順序が異なるだけで，結果的には同一のオブジェクト群が生きているオブジェクトとしてコピーされることになる．

深さ優先順にコピーを行うにはスタックを用いてやればよいが，この GC スタックにどのようなデータを保持するかについてはいくつかのバリエーションが考えられる．ここでは簡潔さのために GC スタックには，Tospace にコピーされたオブジェクトに含まれる「まだコピー先を指していない参照」(「必要だがまだコピーしていないオブジェクト」も表している可能性がある) へのポインタが格納されているものとする．つまり，幅優先順のコピー方式と違い，オブジェクトを Tospace にコピーした際にそれに含まれる参照の位置情報 (アドレス) を GC スタックに push する．

この単純スタック法の基本的な GC 処理としては，GC スタックが空になるまで，GC スタックから pop した位置情報の位置に存在する「Fromspace 内のオブジェクトへの参照」を Tospace を向くように修正しつづけることになる (例えば，図 2 左図での GC スタックのスタックトップはオブジェクト S の第一要素を指しているので，GC スタックから pop してこれを処理すると，図 2 右図のように，オブジェクト S の第一要素が指すオブジェクト T への参照は Tospace 中の T への参照に修正される．その際，T がまだ Tospace にコピーされてなければコピーし，コピーの際には T に含まれる参照の位置情報が GC スタックに push される．図 2 右図では T に含まれる二つ

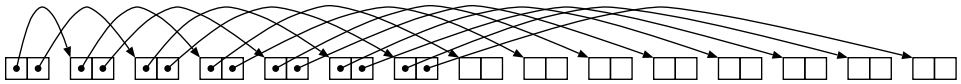

図 3: Breadth-first copying result.

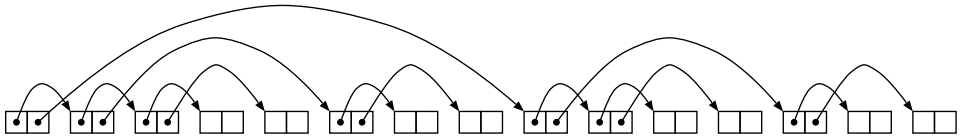

図 4: Depth-first copying result.

の参照の位置情報がプッシュされる .) ここで，オブジェクトの先頭側に位置する参照を先に辿りたい場合は (左優先)，スタックへの push をオブジェクトの後方側から行えばよい．また最後に push した情報はすぐに pop されるので，次に処理する「 Fromspace 内のオブジェクトへの参照 」の位置を保持する変数を準備してやれば一対の push/pop を省略できる．

この単純スタック法の場合，GC スタックに保持するデータは 1 ワード単位であるため個々のスタック操作は高速に行えるが，オブジェクトに多くの参照が含まれる場合は操作数が増えてしまう．また，最悪の場合のスタックの深さは，生きているオブジェクトの数ではなく，生きているオブジェクトに含まれる参照の数に比例してしまう．スタック操作数を減らし，また最悪の場合の深さをオブジェクト数に比例させるには，

- 修正候補である「 Fromspace 内のオブジェクトへの参照 」を保持しているかもしれない Tospace 中のオブジェクトへの参照
- そのオブジェクト内の次の修正候補を得るためのデータ

の組 (2 ワード) を単位としてスタック操作を準備する必要があるが，今回は 1 ワード単位版の実装のみを行った．

3.2 コピー GC におけるメモリアクセスの局所性とオブジェクトの再配置

幅優先順でコピーする場合と，深さ優先順でコピーする場合との違いを，高さ 4 の均一な二分木を例に考えてみる．木のルートからコピーを開始し，子ノードへの参照を辿りながら左詰でコピーしていったときの，コピー後のオブジェクトの配置を，幅優先順，深さ優先順それぞれについて図 3，図 4 に示す．

幅優先順の場合は深さ優先順と比較して親子間の距離が次第に離れていくこと，逆に深さ優先順の場合は木のリーフ付近では親子がすぐ近くに配置されることなどが分かる．また，このような木は，多くの場合，ユーザプログラムでの再帰呼出しによって生成されるが，再帰呼出しによって生成した木はちょうど図 4 の深さ優先順でコピーした場合と同様の配置になると考えられる．

ここで，GC 処理自体でのメモリアクセスの局所性と通常の計算時のメモリアクセスの局所性について考えてみる．

まず，GC 処理自体でのメモリアクセスの局所性としては，

(a) オブジェクトのコピーの際の Fromspace からの読み出し

(b) オブジェクトのコピーの際の Tospace への書き込み

(c) コピー先を得るための Fromspace からの読み出し

(d) コピー先を設定するための Fromspace への書き込み

(e) Fromspace へ向いた参照の修正のための Tospace に対する読み書き

を考えてやる必要がある．(b) については，空き領域の先頭を連続的に用いるので幅優先順でも深さ優先順でも局所性はよいものと思われる．(e) については，幅優先順の場合は連続的にアクセスされ，深さ優先の場合は最近 (b) でコピーしたばかりのオブジェクトへのアクセスとなるため局所性はよいものと思われる．残った問題は (a)，(c)，(d) の Fromspace へのアクセスの局所性である．これらはまず，(c) のアクセスが行われた後，未コピーであれば (a) と (d) のアクセスが連続して実行される．よって Fromspace 内のオブジェクトの配置に適合するようなパターンでアクセスする場合には局所性が高まる．つまり，Fromspace 内のオブジェクト配置が再帰呼出しによって生成した木の形ならば，幅優先順よりも深さ優先順でアクセスしたほうが局所性が高いと考えられる．実際，5 節の予備実験では，幅優先順と深さ優先順で GC 処理時間に開きがあり，その主な要因は Fromspace 内のオブジェクトの配置であると考えられる．

通常の計算時のメモリアクセスの局所性について考えてみる．通常の計算時には，

(a) 新しく生成するオブジェクトへのアクセス

(b) データ構造の参照を辿る向きのアクセス

(c) 関数フレーム等に保存してあった参照を辿るアクセス

が考えられる．(a) については，連続した空きメモリ領域の部分に生成されるため局所性は比較的よいと考えられる．(c) については以前アクセスしていたオブジェクトへ (木構造でいえば親のオブジェクトへ) 戻るようなアクセスであり，そのようなオブジェクトは比較的近い過去にアクセスされていることが多く，局所性はよいと考えられる．残った問題は (b) のアクセスである．該当するオブジェクトへのアクセスの局所性をよくするには比較的近い過去にアクセスされているものの近くにそのオブジェクトが配置されている必要がある．すると，少なくとも親オブジェクトのそばに配置をするという観点からいえば，図 3，図 4 からも分かるように，幅優先順コピー GC よりも深さ優先順コピー GC のほうが有利である．幅優先の場合は兄弟オブジェクトや従兄弟オブジェクトが近くにいるが，直前にそれらにアクセスしていないようなアクセスでは局所性はよくならない．5 節の予備実験では，木全体をトラバースするようなユーザプログラムとしているため，直前に兄弟，従兄弟へのアクセスがなされていることが多い．このため，5 節の予備実験では，通常計算の実行時間には幅優先順コピー GC と深さ優先順コピー GC でほとんど差は見られなかった．

3.3 余分なスタック領域を用いない深さ優先順コピー方式

コピー GC 処理では，ルートから到達可能なオブジェクトを残さずコピーしなくてはならず，また，コピー後のオブジェクトはお互いにコピー後のオブジェクトへの参照を持たなくてはならない．よって「生きているがまだコピーしていないオブジェクト」を忘れないためにも「まだコピー先を指していない参照」について残さず調査していく必要がある．幅優先順の場合は，調査範囲を Tospace 中の s から b までの範囲としてスキャンしていくことができた．つまり，すべての調査対象のデータを保持しなくても調査対象は二つのポインタの間に連続的に存在するということだけ管理しておけばよかった．一方，深さ優先順の場合は，調査対象のデータを別途スタック等を用いて管理しなくてはならない．仮に，幅優先順の場合と同じように，Tospace 中のある範囲に調査対象が存在するということだけを管理するとしたら，調査対象は Tospace にコピーされたオブジェクトの占める範囲全域に渡って存在するので，最新の b から Tospace の先頭に向かって (すでに調査したオブジェクトも含めて) 何度もスキャンし直すという方法になってしまう．そのようなスキャンに必要なトータルの時間は，すべての生きているオブジェクトが持つ参照の個数

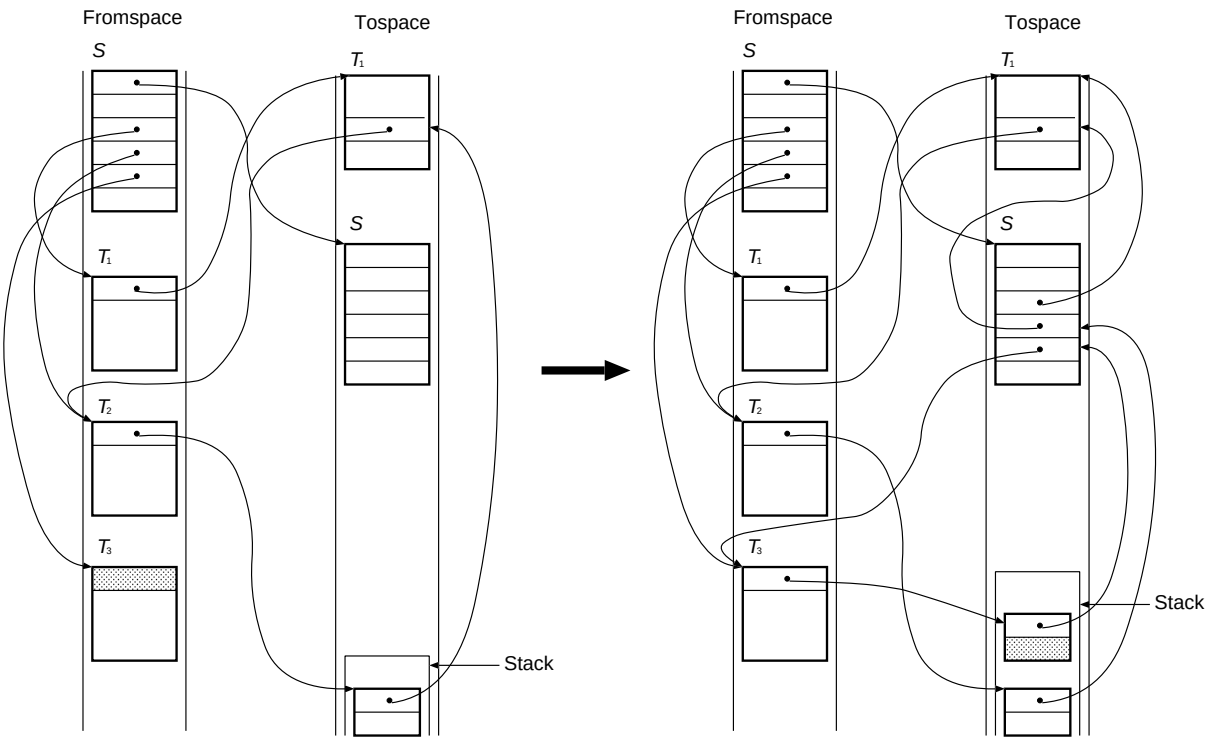

図 5: Depth-first copying collection with reservation stack.
(T_1 から T_2 への参照と、S から T_1、T_2、T_3 への参照)

の二乗のオーダーとなってしまうので現実的ではない．これを避けるためには次の調査対象が定数オーダーで見つけられる必要がある．そのためには，逆転ポインタ法などのスタック以外の方式でも構わないが，少なくともスタックが保持する情報と同等の情報を保持できるような方式が必要である．

単純にスタックを用いて深さ優先順にコピーする方式の場合，そのために必要なスタックの深さは，最悪の場合，生きているオブジェクトの個数に比例する．コピー方式で 2 倍のメモリを必要とすることに加えて，ヒープサイズに比例する余分なスタック領域を必要とするのは好ましくない．

これを改良し，予約スタックという特殊なスタックを Tospace の末端に配置することで余分なスタック領域を必要としない方式 [中島 95] が提案されている．この方式では，(1) スタックに位置情報を push するのはその位置から参照されるオブジェクトが「参照を二つ以上含み，かつ，未コピーの場合」のみとし，かつ，(2) すでにスタックに push されている位置情報に位置する参照と同じ参照の持つ位置の情報はスタックの深さを増やすことなく push できるようにしている (図 5)．

(1) については push する前にそのことを確認する．例えば図 5 においてオブジェクト S の持つオブジェクト T_1 への参照は既にコピーされた T_1 が存在しているため，オブジェクト S 内の位置情報を push することなしにコピーされた T_1 を参照するように修正を行う．また，T_1 などのオブジェクトが参照を一つまでしか含まない場合はその先の処理を繰り返して行うことでスタックを使わなくて済む．

(2) のために，既に push されているスタックの要素があるかどうかが判定できる必要がある．そこで，例えば，オブジェクト T_3 への参照を S は保持しているが，オブジェクト S 内でその参照が位置する位置情報をスタックに push する際に T_3 の先頭の要素も待避し (図 5 のグレー部分)，T_3 の先頭要素には予約スタックの要素へのポインタを上書きする．つまり，T_3 の先頭要素が予約スタック内のアドレスであれば既に push されていると判断できる．さらにスタックに実際に push するのは位置情報ではなく，位置情報そのものをリンクで結んだリンクトリストの先頭の情報と

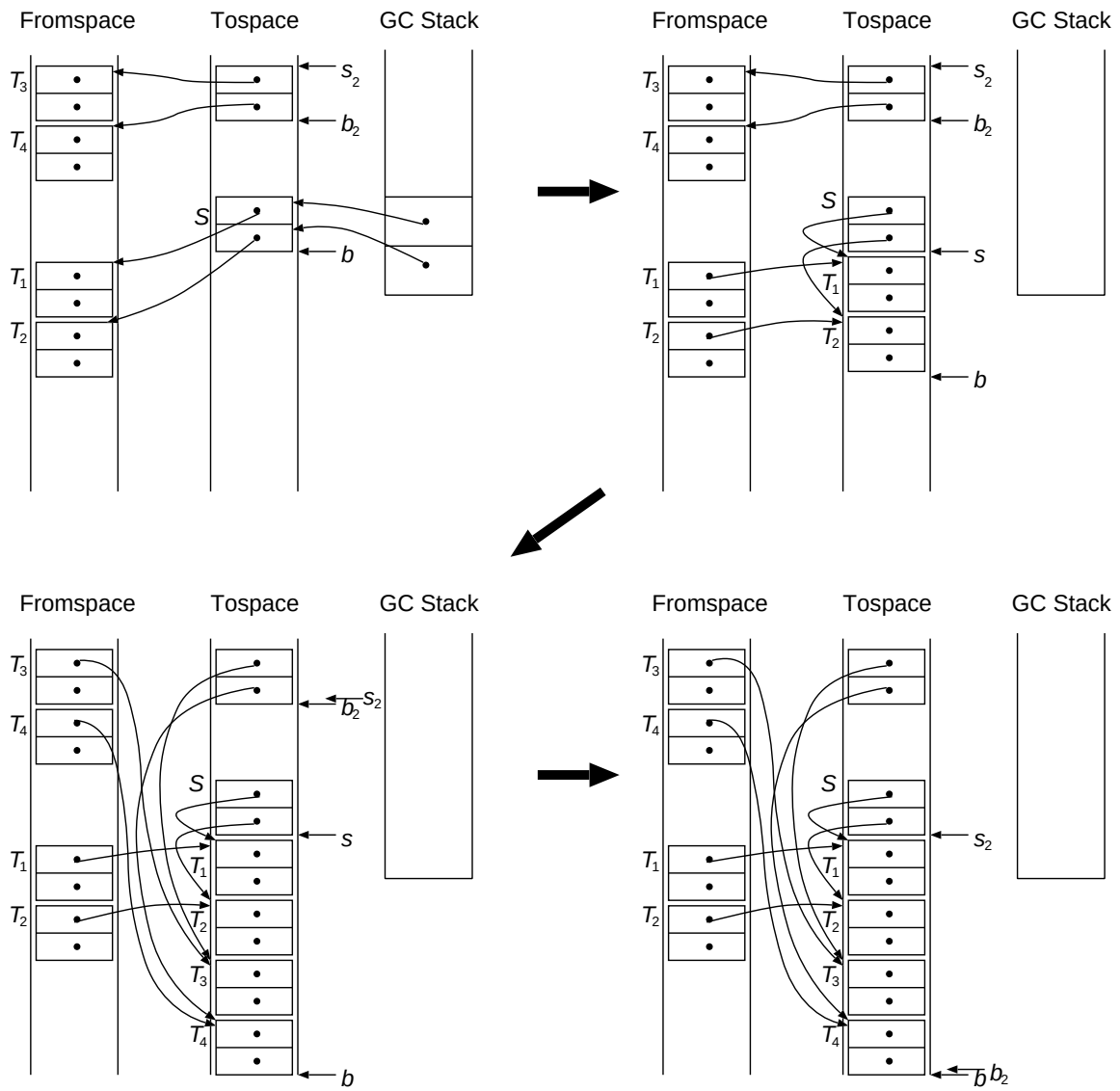

図 6: Mostly-depth-first copying collection with small stack space.
(S から T_1、T_2 への参照)

する．このため例えば，S が保持するオブジェクト T_2 への参照の位置はそのリンクの先頭に挿入している．

予約スタックの深さは「参照を二つ以上含み (つまり 2 ワード以上の) かつ未コピーのオブジェクト」の数よりも大きくなることはないため，予約スタックの先頭と Tospace の b が衝突しないことが保証される．

予約スタック法では，深さ優先順のコピーを従来の幅優先順のコピー方式と同じサイズのメモリ領域で実現しているという優れた特長を持つが，予約スタックの管理には実行コストの高い複雑な処理が必要となり処理性能に難点がある．

4　大部分を深さ優先順にコピーするゴミ集め方式

提案する少量のスタックで大部分を深さ優先順にコピーするゴミ集め方式の基本的なアイデアは，既に述べたように，スタックを用いてできるだけ深さ優先順にコピーを行うが，万一スタック

の深さが許される限度を超えるような場合にはスタックを一度空にして深さ優先順のコピーが再開できるように (幅優先順コピー方式と同様に) コピー先の領域のある範囲をキューとして利用するというものである . スタックの最大サイズが限定されているため提案する方式を限定スタック法と呼ぶことにする .

似たようなアイデアは [Che70] の中で , 幅優先順にコピーする GC 方式を部分的に深さ優先順とする方式として述べられている . これは幅優先順にコピーする GC 方式において , 一つのオブジェクトをコピーする際に , そのオブジェクトから参照される未コピーのオブジェクトが一つでもあれば , そのオブジェクトもコピーし , さらにそのオブジェクトから参照される未コピーのオブジェクトについてもこれを繰り返すというものである . これは単なる繰り返しなのでスタックは不要となる .

提案する限定スタック方式と [Che70] のスタックを用いない semi-depth-first 方式との違いは , 限定スタック法ではスタックによる深さ優先順のコピーに支障がある場合にのみ , 一部に幅優先順のコピー方式を利用するが , 逆に [Che70] では幅優先順のコピー方式においてスタックを使わずに済ませられる繰り返しの範囲内で一部に深さ優先順のコピーを利用する点といえる .

提案する限定スタック法を図 6 の左側に示す . Fromspace , Tospace や , Tospace の空き領域の先頭を指しているポインタ (図 6 の b) や , GC スタックについては深さ優先順の単純スタック法の場合と同様である .

限定スタック法では単純スタック法の場合に加えて幅優先キューに相当する未スキャン領域の先頭 (図 6 の s_2) と未スキャン領域の末尾 (図 6 の b_2) を指すポインタを用いる . s_2 と b_2 に挟まれた範囲が「まだコピー先を指していない参照」(「必要だがまだコピーしていないオブジェクト」も表している可能性がある) の集合を管理するためのキューとして利用されるという点は幅優先順のコピー方式における s と b に挟まれた範囲と同様である . 一方 , このキュー以外に GC スタックにも「まだコピー先を指していない参照」の位置情報が保持されている点と , コピーは b を更新しながら行うためスタックを利用して深さ優先順にコピーを行っている間は b_2 は変化しないという点が異なる .

アルゴリズムの基本部分は単純スタック法と同じである . すなわち , GC スタックから pop したアドレスにある参照が「Fromspace 内のオブジェクトへの参照」であれば , 修正を行うとともに , Tospace にオブジェクトをコピーしたときにはコピーしたオブジェクトに含まれる参照の位置情報を GC スタックに push する . ただし , スタックの深さがあらかじめ限定しておいたサイズを超えるような場合は , s_2 と b_2 に挟まれた範囲にのみ連続して GC の調査対象となる「まだコピー先を指していない参照」が存在するように一連のコピーを行うことで (つまり図 6 の右下の状態まで遷移することで), スタックを一旦空にする .

図 6 の遷移を順に見ていく . まずそのときの b が指す位置を覚えておく (以下では s によって指されるものとしよう). スタックを一旦空にするには単純スタック法の基本的な GC 処理と同様に , GC スタックが空になるまで , GC スタックから pop した位置情報の位置に存在する「Fromspace 内のオブジェクトへの参照」を Tospace を向くように修正しつづけることになる . ただし , このときオブジェクトを Tospace にコピーした際でも , それに含まれる参照の位置情報を GC スタックに push することは中止する . 新たな push は中止されているのでスタックを着実に空にすることができる (図 6 の左上から右上への遷移 .)

この時点で s_2 と b_2 に挟まれた範囲 , s と b に挟まれた範囲の二つのキューにより「まだコピー先を指していない参照」が管理されている . 次に s_2 を b_2 までスキャンさせて s と b に挟まれた範囲のキューにオブジェクトをコピーしていく (図 6 の右上から左下への遷移 .)

s_2 と b_2 に挟まれた範囲がなくなったら s_2 と b_2 のポインタの値は不要となり ,「まだコピー先を指していない参照」は s と b に挟まれた範囲にのみ存在するので , この時点の s と b のポインタ値をそれぞれ s_2 と b_2 に再設定すれば深さ優先順のコピーを再開することができる (図 6 の右下への遷移).

つまり , スタック上のデータを元にしたコピーと , s_2 と b_2 間のキューのデータを元にしたコピーを連続して行うことで , スタックと s_2 と b_2 間のキューを共に空にして , 新しい一つのキューにの

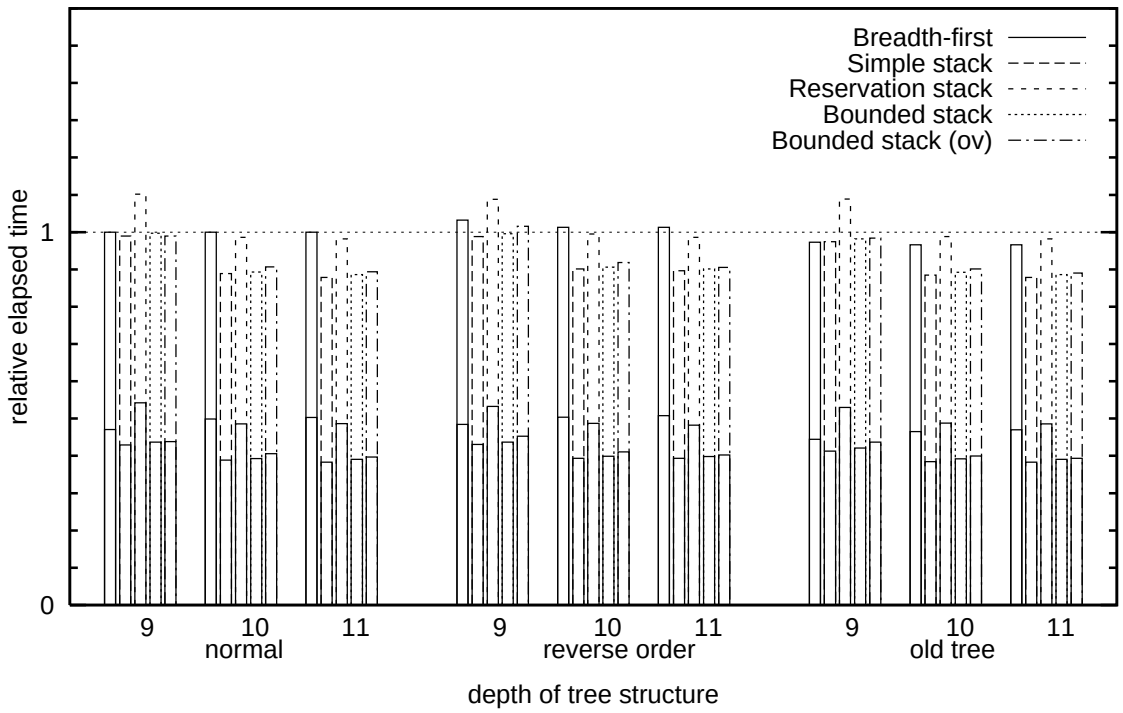

図 7: Comparison of GC methods

Breadth-first(幅優先)，Simple stack(単純スタック法)，Reservation stack(予約スタック法)，Bounded stack(限定スタック法) の性能比較．横軸の木の深さ $d(=9,10,11)$ に対し，縦軸は約 1.5×3^d 個のオブジェクト (一個当り 80 バイトの領域を使用) からなる木のコピーを 20 回繰り返すユーザプログラムの幅優先法の実行時間を 1 としたときの実行時間 (棒グラフの内側の棒は GC の時間)，Tospace のサイズは 5MB, 15MB, 45MB とした．「Bounded stack (ov)」は限定スタック法でのスタックサイズを故意に小さく取ってスタックが満杯になるケースを引き起こしたものである．

み調査すべき参照が存在する形にしたのである．さらにいえば，複数個のスタックやキューを空にしつつ，それらのスタックやキューのデータを元に新しい一つのキューだけが残るようにするといった発展形も考えられる．

GC スタックが空になったときは，s_2 と b_2 に挟まれた範囲をキューとして s_2 を動かして得られた参照から深さ優先順にコピーを行うようにする．GC スタックが空になり，かつ，s_2 が b_2 に追い付いており，かつ，すべてのルートの処理が終わったら，コピーは完了である．

限定スタック法では，スタックの最大サイズを 128 バイト程度に限定しても (32bits のポインタなら) 32 個までスタックの要素を保持でき，この程度の要素数があればスタックを使いきることなくコピーが完了する場合は多い．また，仮にスタックがあふれた場合も一階層分だけ幅優先順でコピーすれば良く，その後は深さ優先順でのコピーを再開できる．また，深さ優先順でのコピー中は単純スタック法と比べてスタックオーバーフローのチェックのわずかなコストを追加するだけでよいので高速な処理が実現できる．

5 予備実験

従来の幅優先順コピー，単純スタック法による深さ優先順コピー，予約スタック法による深さ優先順コピー，限定スタック法による準深さ優先順コピーの性能比較に関する予備実験の結果を図 7 に示す．図 7 は 4 つの方式それぞれについて幅優先法の実行時間を 1 としたときの実行時間

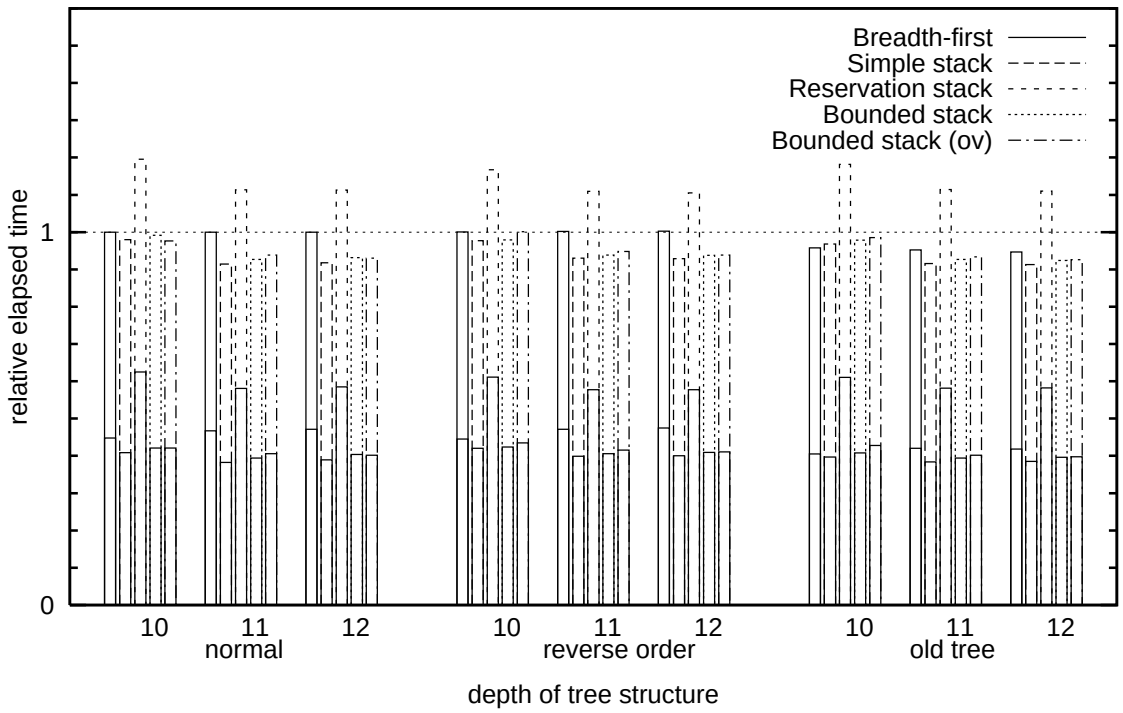

図 8: Comparison of GC methods for small objects

横軸の木の深さ $d(=10,11,12)$ に対し，縦軸は約 1.5×3^d 個のオブジェクト (一個当り 24 バイトの領域を使用) からなる木のコピーを繰り返すユーザプログラムの相対実行時間，Tospace のサイズは 4.5MB, 13.5MB, 40.5MB とした．

を示している．

評価環境には，Sun Ultra 30(UltraSPARC-II 296MHz，オンチップキャッシュ32KB，外部キャッシュ 2MB，主記憶 128MB，Solaris 2.6) を用いた．コピー GC 本体とユーザプログラムは C 言語で記述し GNU gcc 2.8.1， 最適化 O4 によりコンパイルして実験を行った．評価に用いたプログラムはフラクタル図形をあるレベルまで近似した木構造の図形データを生成したあと，古い木構造への参照を消しながら，木構造を繰り返しコピーするものである．このコピーは，GC ではなくユーザプログラムによる深さ優先のコピーのことである．測定は木構造のコピーの部分について，GC が処理を行った時間とトータルの実行時間を測定した．図 7 の棒グラフの内側の棒は GC の時間を表している．図 7 のグラフの横軸は木の深さ $d(=9,10,11)$ であり，木構造は 3 分木であるため約 1.5×3^d 個のオブジェクト (一個当り 80 バイトの領域を使用) からなる．$d(=9,10,11)$ それぞれについて Tospace のサイズは 5MB, 15MB, 45MB とした．

図 7 の左側のグループ (normal) では特に何もしなかった場合の結果を示している．木の深さが 9 でメモリ領域が 5MB のときには，予約スタック法が遅い以外はほとんど性能差はみられなかった．しかし，木の深さが 10 でメモリ領域が 15MB のときには，幅優先順コピー方式，予約スタック法と比べて，単純スタック法，限定スタック法は良い性能を示している．その差は木の深さ 11 でメモリ領域が 45MB のときにも同様であった．さらに木の深さ等を大きくすることに関しては，主記憶が 128MB であるため，木の深さ 12 の場合はページアウトが頻発してプログラムの実行が困難であった．限定スタック法にすることによる性能向上は，GC 処理部分については最大 22%程度処理時間を短縮し，ユーザプログラムと GC を合わせた実行時間については，最大 11%程度の短縮であった．

また，評価に用いたプログラムではスタックのオーバーフローも生じないため，単純スタック法と限定スタック法の性能差はほとんどない．$d(=9,10,11)$ それぞれについて限定スタック法の

スタックの深さの限度を 14，16，18 と故意に小さく取ってスタックが満杯なるケースを引き起こしたものを図 7 では「Bounded stack (ov)」として示した．このときスタックは 1 回の GC 当り 7 回満杯になっているが，これについても単純スタック法との性能差はほとんどない．

図 7 の真ん中のグループ (reverse order) では，ユーザプログラムが行う深さ優先順のコピーと，コピー GC が行う幅優先順または深さ優先順のコピーとの間で，左右の優先順を逆にしたものである．つまり，ユーザプログラムでは木の左側を優先して深さ優先順にコピーし，コピー GC では木の右側を優先して深さ優先順か幅優先順 (実は幅優先順とは浅さ優先順でもある) にコピーを行う．この食い違いのため局所性が低下するものと考えられる．実際には全体的に若干遅くなる程度であった．これは，最終的に木のリーフの部分では向きが逆とはいえ連続的なアクセスがなされるためだと思われる．

図 7 の右側のグループ (old tree) では，ユーザプログラムで行うコピーの後，コピー後の木ではなくコピー元の木のほうを残すようにユーザプログラムを変更したものである．木をコピー GC でコピーする際にはユーザプログラムが生成した順序とは異なる順序でコピーがなされ，その食い違いが性能低下となって現れると考えられるが，コピー元の木のほうを残すことで，次回の GC におけるコピーでは一つ前の GC 処理でコピーした順に木のオブジェクトは配置されている．このため二回目以降の GC では古い木に関してはアクセスの局所性は高まっているはずである．実際，ユーザプログラムとの食い違いがある幅優先法について特に実行時間が短縮されている．

図 8 には，上記のフラクタル図形のユーザプログラムから浮動小数点数に関する演算やオブジェクトの浮動小数点数のフィールドを取り除いたものについて評価を行った結果を示す．ほぼ同様のプログラムではあるが，木の深さ $d(= 10, 11, 12)$ に対し，約 1.5×3^d 個のオブジェクト (一個当り 24 バイトの領域を使用) からなる木のコピーを行う．Tospace のサイズは 4.5MB, 13.5MB, 40.5MB とした．図 8 では，図 7 と比較して，いつかの点が強く現れている．例えば，予約スタック法の処理速度の問題はより顕著であり，古い木を残す場合の古い木に関するアクセスの局所性の向上の効果は，特に幅優先法において実行時間の短縮として強く現れている．

6 関連研究

Moon の approximately depth-first algorithm[Moo84] では，幅優先順のコピー GC の枠組みのなかで，メモリアクセスの局所性を高めるために部分的に深さ優先順的なコピーとなる工夫を追加している．このため，幅優先順 GC で使用される，未スキャン領域の先頭の s と，空き領域の先頭かつ未スキャン領域の末尾である b 以外に，もう一つ未スキャン領域の一部の指すポインタ p を導入する．そして，s と b の間のスキャンより優先的に，p と b の間のスキャンを行う．p は s と b の間にとる．常にスキャン領域の本当の先頭は s が保持しているので，p は比較的自由に設定することができるが，後戻りを避け，p は必ず b に近づく方向にしか動かないものとする．さて，見つかった未コピーのオブジェクトは b の位置にコピーされるので，メモリアクセスの局所性を高めるには，次にコピーされるオブジェクトの参照元は b と同じページにあることが望ましい．よって b が別のページまで移動してしまったときは，p を b と同じページの先頭に再設定する．p が b に追い付いてしまったときには，$p-b$ 間でスキャンを続けるための種を準備するために s からのスキャンを少し行う．s は，p によるスキャンが済んでいる範囲をもう一回スキャンする可能性があるが，その手間は仕方がないものとする．つまり，Tospace の一部は，2 回スキャンされる可能性がある．

Moon の方式は，参照元と参照先のオブジェクトをできるだけ同一のページに置くということを目指している一方で，それぞれのページの内部については幅優先順のコピーが行われる．つまり，それぞれのページ内部は図 3 のような配置となる．このため，仮想記憶の面からは局所性に優れているが，キャッシュの面からは局所性はよくならないと考えられる．5 節の予備実験における深さ優先順 GC の性能向上は主にキャッシュに関する局所性の向上に起因すると考えられ，Moon の方式をそのまま用いたのでは，同様の性能向上は達成できないものと考えられる．

Wilson らによる hierarchical decomposition [WLM91] では，Moon の方式とほぼ同一のオブジェクトの配置 (ページ単位で順序が入れ替わっている程度の違いしかない) が得られる．Wilson らのアルゴリズムでは，各ページごとにローカルに未スキャン領域を管理するためのポインタを二つずつ用意することで，Moon の方式で問題となった再スキャンを避けることができる．アルゴリズムとしては，2 レベルの Cheney アルゴリズムとなっており，ヒープ全体の幅優先順コピー GC の上で，各ページ内の幅優先順コピー GC を優先的に行うものである．Wilson らの方式もページ内は幅優先順のコピーとなっているためキャッシュに関する局所性については Moon の方式と同じことがいえる．

Wilson らはまた，hierarchical decomposition で得られた配置について，

1. 木のルートの付近は index の役目を持つ重要な部分であり，それらを同じページにまとめることは，木のどの部分をアクセスするにしても重要となる．
2. あるオブジェクトから参照されるどの子オブジェクトへのアクセスを行ってもそれらが同じページにあるような配置となっている (一方，深さ優先の場合は左の子は近くに位置するが右の子は別ページに位置するといったことが起り得る .)

という二つの利点を挙げている．ページに関するこれらの局所性は確かに重要であり，キャッシュに関する局所性の良さを保ったままこれらの利点を取り入れていく必要があると考えられる．

7　おわりに

本研究では，深さを限定した少量のスタックを用いて大部分のオブジェクトを深さ優先順にコピーするゴミ集め方式を提案した．提案方式は高速な処理を特長とし，128 バイト程度のスタックを追加することでメモリアクセスの局所性を改善することができる．

現在，より詳細な評価結果を得るためのさらに詳しい測定を進めている．木構造以外に双方向リストなどを試みると興味深いと考えている．

提案するゴミ集め方式を並列言語に組み込み並列計算機で処理できるようにすることも検討している．並列計算機では優れた負荷分散が重要となるが，提案する方式は深さ優先順をベースとするため比較的ルートに近い情報を用いて授受する負荷の粒度を大きく保つことが可能であると考えている．

参考文献

[Che70] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, Vol. 13, No. 11, pp. 677–678, November 1970.

[Moo84] David A. Moon. Garbage Collection in a Large Lisp System. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pp. 235–246, August 1984.

[WLM91] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective “Static-graph” Reorganization to Improve Locality in Garbage-Collected Systems. *ACM SIGPLAN Notices*, Vol. 26, No. 6 (Proceedings of PLDI'91), pp. 177–191, June 1991.

[中島 95] 中島浩, 近山隆. スタック領域が不要な深さ優先順コピー型ゴミ集め方式. 情報処理学会論文誌, Vol. 36, No. 3, pp. 697–713, March 1995.