階層的グループ化に基づくコピー型ごみ集めによる局所性改善

八杉昌宏[†] 伊藤智一[†]
小宮常康[†,☆] 湯淺太一[†]

プロセッサとメモリの性能差は近年拡大しており，仮想記憶の局所性だけでなく，キャッシュの局所性を改善することも重要となっている．ごみ集め（GC）の 1 つであるコピー GC では，2 つの部分空間の片方から生きているオブジェクトのみを他方の部分空間にコピー（移動）する．Cheney の非再帰的なコピー GC は，コピー先の部分空間の一部をスキャン待ちオブジェクトのキューとして利用するので追加の作業領域をほとんど必要としないが，オブジェクトが幅優先順にコピーされる．しかし幅優先順のコピーでは，多くの応用プログラムでメモリアクセスの空間的局所性が低下し，キャッシュミスや仮想記憶のミス（ページフォールトや TLB ミス）が増加する．本研究では，できるだけ深さ優先順にコピーする方式を提案する．実際にはそれでも局所性の改善は限定的なため，さらに，オブジェクトを階層的にグループ化してコピーする方式を提案する．これらの再帰的なコピー GC ではコピーの際に使用するスタックなどの作業領域のサイズが最悪の場合に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る．そこで，限られた作業領域でできるだけ多くの部分に深さ優先コピーや階層的グループ化を高速に適用する方式を提案する．本論文では提案手法によるミス率低減効果の測定結果も示す．

Improving Locality by Copying Garbage Collection Based on Hierarchical Clustering

Masahiro Yasugi,[†] Tomokazu Ito,[†] Tsuneyasu Komiya[†,☆]
and Taiichi Yuasa[†]

The increasing processor-memory performance gap makes improving cache locality as important as virtual memory locality. Copying garbage collection moves all live objects from one semi-space to another semi-space. Cheney's nonrecursive copying algorithm employs a section of the destination semi-space as a queue of unscanned objects and requires only a negligible working space, but objects are copied in breadth-first order. However, in many applications, the breadth-first copying makes the spatial memory access locality worse and increases cache misses, page faults and TLB misses. In this paper, we propose a mostly depth-first copying algorithm. Since depth-first copying only achieves limited locality improvement, we also propose a new "hierarchical clustering" copying algorithm. Since these recursive copying algorithms require a working space (e.g., a stack) for recording unscanned objects whose worst-case size is proportional to the number of live objects, we also propose an efficient technique which achieves depth-first copying or hierarchical clustering as approximately as possible using bounded workspace. This paper also shows the effect of our approach on miss-rate reduction.

1. はじめに

自動的なメモリ領域管理を行うごみ集め（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良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 1 つであるコピー方式では，今後アクセスされる可能性のある（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（ごみ）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

† 京都大学大学院情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University
☆ 現在，豊橋技術科学大学情報工学系
Presently with Department of Information and Computer Sciences, Toyohashi University of Technology

じサイズで 2 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．2 つの部分空間の役割は GC のたび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付けが高速化できる．

Cheney によって考案された代表的なコピー GC[1] では，コピー先の部分空間の一部をキューとして利用するのでオブジェクトは幅優先順にコピーされる．この方式では，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，それ以上の追加のメモリ領域をほとんど必要としない．しかし，幅優先順のコピーでは，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，応用プログラムでは，多くの場合，オブジェクトへの参照を幅優先順にたどることはないためである．

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[6] 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．そこで，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，限られた作業領域（たとえば 128 バイト程度のスタック）でできるだけ深さ優先順にコピーしてメモリアクセスの局所性を改善する方式（限定スタック法）[7] を提案する．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．

しかし，限定スタック法も場合によっては局所性の改善は限定的だった．そこで，オブジェクトを階層的にグループ化してコピーすることで局所性を向上させる方式も提案する．階層的グループ化に基づくコピー型ごみ集め方式でも，コピーの際に使用する作業領域は最悪の場合，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る．そこで，限られた作業領域でできるだけ多くの部分に階層的グループ化を高速に適用する方式を提案する．

本論文では，2 章でプログラム実行の高速化の点からメモリアクセスの局所性について議論する．3 章では従来の幅優先となるコピー GC など，スタックを用いない非再帰的なコピー GC について説明する．4 章ではコピー GC 方式の違いによるメモリアクセスの局所性について議論する．5 章で再帰的なコピー GC 方式について述べ，限定スタック深さ優先コピー GC や階層的グループ化コピー GC などの方式を提案する．またミス率低減効果を含む測定結果を 6 章で示す．

2. メモリアクセスの局所性と高速化

プログラムの実行の高速化のために，仮想記憶（オペレーティングシステム）とキャッシュ（ハードウェア）の観点からどのようにメモリアクセスの局所性を改善したらよいかについて議論する．

2.1 仮想記憶とメモリアクセスの局所性

仮想記憶方式で，オペレーティングシステム（OS）はプロセスの進行に必要なメモリを管理できる．ユーザプログラムでのメモリアクセスには論理アドレスを利用させる．ユーザプログラムが論理アドレスでメモリアクセスしようとすると，OS が管理下のメモリ管理ユニット（MMU）というハードウェアで論理アドレスから物理アドレスにアドレス変換して物理メモリのデータにアクセスするようにする．アドレス変換はたとえば 8 KB のページを単位として論理アドレスをインデックスとして変換表（ページテーブル）を用いればよい．MMU のアドレス変換が順調にできるほとんどの場合にはユーザプログラムから OS にトラップはしないようにする．MMU は OS からは利用できるが，ユーザプログラムは MMU が見せる仮想記憶のイメージが見えるのみである．

また，よく知られているように仮想記憶方式では物理メモリ容量より大きな仮想アドレス空間を用いてよい．この場合，対応する物理アドレスを持たないページのデータはディスクなどに保存しておき，そのページの論理アドレスに対するアクセスがあった場合はアドレス変換ができないとして，トラップ（ページフォールト）する．ページフォールトが発生すると，OS は，空きがなければあまり利用されていないページを決めて，その物理メモリのデータのディスクへの書き出し（ページアウト）を行った後，必要なページのデータのディスクからの読み込み（ページイン）と，ページ

テーブルの更新を行い，トラップからの復帰が可能なようにする．さらに MMU では，変換を高速化するために変換表のよく使う部分をキャッシュしたような Translation Lookaside Buffer（TLB）を備えるのが普通である．TLB で変換できなかった場合には **TLB ミス**となるが，自動的にページテーブルから更新するもののほかに，たとえば UltraSPARC ベースの計算機のように TLB ミスで OS にトラップするような計算機もある．後者の場合はページフォールトはトラップではなく，TLB ミスによるトラップの処理に含まれることになる．

ページフォールトが頻発するようなプログラムの実行速度は劇的に低下する．これは,（物理）メモリアクセスの速度とディスクアクセスの速度差による．このため，物理メモリ容量を超えるページ数を必要とするようなワーキングセット（プログラムのある時点でよくアクセスのアドレスの集合）とならないように工夫する必要がある．

また，TLB ミスのペナルティも実は大きく，TLB ミスを少なくするのも，大雑把にいえばワーキングセットを小さくすればよい．他のいい方では，メモリアクセスの局所性を良くすればよい．メモリアクセスの局所性には，**時間的局所性**（最近アドレス x にアクセスしているなら，今，同じ x にアクセスする可能性は高い）と**空間的局所性**（最近アドレス x にアクセスしているなら，今，その近くの x' にアクセスする可能性は高い）がある．もうすこし具体的な方法としては，プログラムの計算手順に順序を入れ替えてよい操作（の集まり）があれば，同じページにできるだけ時間的に集中してアクセスするような計算順序に修正するとよい．たとえば，S_k をアドレスが $ak+b$ 付近のメモリに多くアクセスする操作の集合として，$S_0 \to S_1 \to \cdots S_n \to S_0 \to S_1 \to \cdots S_n$ のような計算順序でメモリにアクセスするプログラムがあった場合，計算順序を $S_0 \to S_0 \to S_1 \to S_1 \to \cdots S_n \to S_n$ と入れ替えられれば近くのメモリにアクセスする操作集合を時間的に隣り合わせることができ，TLB ミスを減らすことができる．また，処理の時間的な順序ではなくデータの空間的な配置を工夫する方法もある．たとえば 2 次元配列 `a[1000][1000]` に，

```
for(j=0;j<1000;j++)
  for(i=0;i<1000;i++)
    a[i][j] += c;
```

のようにアクセスすると `a[i][j]` と `a[i+1][j]` は離れているため局所性は悪くなる．ここで，計算順序を入れ替えられるならば，

```
for(i=0;i<1000;i++)
  for(j=0;j<1000;j++)
    a[i][j] += c;
```

などとして `a[i][j]` と `a[i][j+1]` は隣り合っているため局所性が良くできる．しかし，なんらかの理由で計算順序を入れ替えられないならば，

```
for(i=0;i<1000;i++)
  for(j=0;j<1000;j++)
    b[j][i] = a[i][j];
```

のように転置行列となるよう 2 次元配列 `b[1000][1000]` にいったんコピーして，`a` の代わりに `b` を用いれば

```
for(j=0;j<1000;j++)
  for(i=0;i<1000;i++)
    b[j][i] += c;
```

のようになり，`b[j][i]` と `b[j][i+1]` は隣り合っているため局所性が改善される（この例では転置自体の局所性は悪いが，後でそれを上回る効果が得られるならば，いったん転置する利点はある）．本論文で扱う局所性向上はこのようなデータの空間的配置の工夫に基づくもので，計算手順の入れ替えに基づくものではない．また，データの空間的配置を，コピー GC で自動的に行おうというものである．

2.2 キャッシュとメモリアクセスの局所性

レジスタアクセスを行うのに比べてメモリアクセスは遅い．高速なプロセッサが次々開発されるのと比較して，メモリアクセスの遅延短縮やバンド幅向上はなかなか進まないでいる．このため命令実行部のできるだけ近くに容量は少なくてもよいので高速にアクセスできるデータの隠し場所（キャッシュ）を設けて，遠くの容量の多いメモリへのアクセスを省略する手法がとられる．キャッシュの状態から遠くにあるメモリの状態まですべてトータルでみて，前節で見てきたソフトウェアからみた計算機のモデルにおける物理メモリが実現される（つまりソフトウェアが単純な物理メモリだと考えているものも，実際のハードウェアではキャッシュや**本当の**物理メモリを組み合わせてそのイメージを仮想的に作り上げている）．ソフトウェアからみた計算機のモデルにはメモリのどの部分がキャッシュされているかどうかといったことは含まれない．

キャッシュについても階層化して，命令実行部に近い最高速・小容量の 1 次キャッシュ，少し離れて遅くなるが容量が増えた 2 次キャッシュなどで構成されていることが多い．キャッシュは，

- キャッシュ容量
- ブロックサイズ

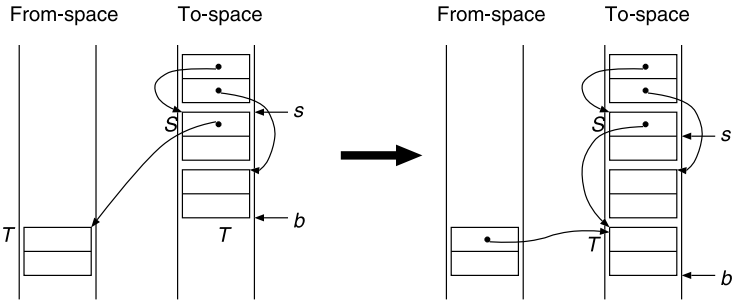

図 1 幅優先コピー GC（S から T への参照を処理）
Fig. 1 Breadth-first copying collection.

- 連想度（連想方式）
- アクセス速度

で特徴付けられる．TLB が通常フルアソシアティブなのに対し，キャッシュの連想度は多くても 8 程度で，ダイレクトマッピング方式（連想度 1）であることも珍しくない．

キャッシュミスを少なくしてプログラムの実行を高速化するには，TLB と同様にメモリアクセスの局所性を良くすればよい．キャッシュミスを少なくすることは高速化に際して非常に高い効果を持つ．TLB ミスの削減に関する議論がある程度成り立つので，以下では，TLB との違いを述べたい．TLB では管理するページが通常 8 KB 程度なのに対し，キャッシュのブロックは通常 64 B 程度である．このため空間的局所性を生かすには同時期にアクセスするデータをしっかり接近させる必要がある．また，連想度が少ないことにも注意が必要である．たとえば，ダイレクトマッピング方式では，キャッシュの容量を S（1 次キャッシュなら 16 KB 程度，2 次キャッシュなら 256 KB 程度）とすると，$Sn + b (n = 0, 1, \ldots, 0 \leq b < S)$ のアドレスのデータは，b で定まるブロックでキャッシュされる．このため S だけ離れた複数のアドレスに頻繁にアクセスする場合，利用するキャッシュブロックの衝突が生じ，キャッシュミスが頻発することになる．キャッシュブロックの衝突を避けるには，頻繁にアクセスされるデータを近付けて，アドレスの差が S 以内になるようにすればよい．つまり，S 以下のサイズのメモリ領域に頻繁にアクセスされるデータを集めればよい．

3. 非再帰的なコピー GC

3.1 Cheney の非再帰的なコピー GC

Cheney の幅優先順にコピーするよく知られた GC 方式（nonrecursive list compacting algorithm）[1] を **図 1** に示す．ヒープの 2 つの部分空間をそれぞれ From-space，To-space と呼ぶことにする．GC を開始する前は，To-space にオブジェクトが生成され（つまり To-space 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），From-space は空であるとする．To-space 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 From-space と To-space の役割を交換し，To-space の空き領域の先頭を指すポインタ（図 1 の b）と，To-space 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，To-space 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 From-space 内のオブジェクトへの参照を To-space 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 To-space のコピーが存在していない場合には，ポインタ b が指す To-space 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，From-space 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 To-space にコピーされたオブジェクト中に存在する「From-space 内のオブジェクトへの参照」についても To-space 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 To-space 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 1 要素がスキャンされ，図 1 右図のように To-space 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」（「生きている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 2 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（生きている）オブジェクトはすべて To-space にコピーされている．この時点で，From-space 内のオブジェクトはすべてごみなので From-space は全体が空き領域となり，To-space 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．

3.2 グループ化を行う非再帰的なコピー GC

非再帰的なコピー GC 時に，**グループ化**（クラスタリング）を行うことができる．関連するデータ（オブジェクト）を集めて隣接するようにメモリに配置するグループ化を行えば，メモリアクセスの空間的局所性を向上させることができる．グループ化にはページサイズのグループ化，キャッシュブロックサイズのグループ化が考えられる．この場合，ページサイズやキャッシュブロックサイズなどが既知でなければならない．

Moon の approximately depth-first algorithm [3] 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．結果としてページ単位でグループ化が行われる．この方式では，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 1 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．つね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 1 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，To-space 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．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それほど良くならないと考えられる．

Wilson らによる hierarchical decomposition [5] 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 2 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 の方式と同様のことがいえる（ただし，キャッシュ衝突ミス回避のためキャッシュ容量（たとえば 256 KB）以下のサイズのメモリ領域に頻繁にアクセスされるデータを集めるという点では，Moon の方式より有利と思われる）．

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うる），

という 2 つの利点をあげている．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改良でも，これらの利点を取り入れていく必要がある．

GC ではないが，キャッシュに関するグループ化を行うものには Chilimbi らの cache-concious structure layout [2] がある．この方式では，均質なオブジェクトをノードとする木のようなデータ構造について，ノー

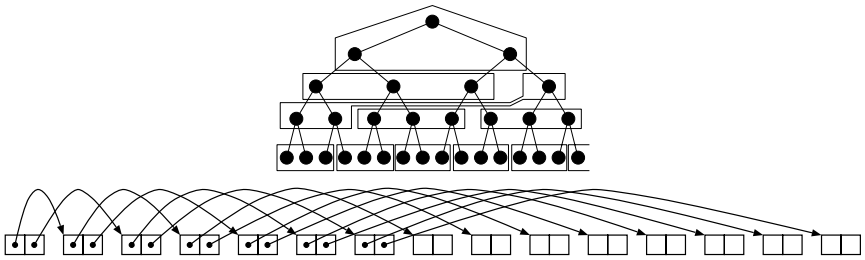

図 2 幅優先方式の結果
Fig. 2 Breadth-first copying result.

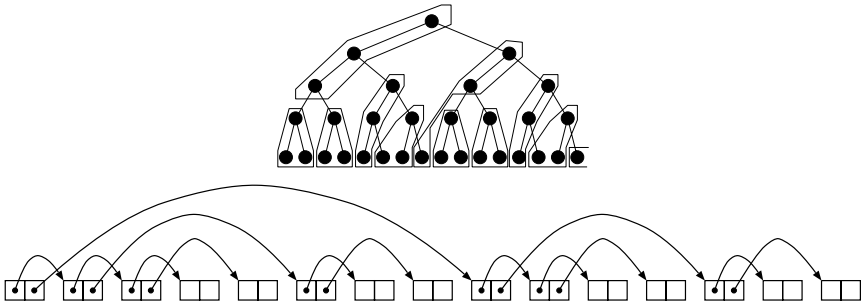

図 3 深さ優先方式の結果
Fig. 3 Depth-first copying result.

ドとその子ノードをグループ化してコピーしたり，木のルートの近くのような頻繁にアクセスされる部分について衝突によるキャッシュミスを避けられるようグループ化してコピーしたりする．キャッシュの容量や，連想度，ブロックサイズなどをパラメータとしてデータ構造変換を行うことを提案している．この結果，2 分探索木による評価で最大で 40%程度の高速化を実現している．

4. コピー GC とメモリアクセスの局所性

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たどる向きのアクセス
(c) 関数フレームなどに保存してあった参照をたどるアクセス

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良いと考えられる．(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，局所性は良いと考えられる．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良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．そのためには，親オブジェクトの近くに配置をするのが有効といえる．

高さ 5 の均一な 2 分木を幅優先，深さ優先でコピーしたときの結果をそれぞれ**図 2**，**図 3** の上部に示す．また，高さ 4 の均一な 2 分木について木のルートからコピーを開始し，子ノードへの参照をたどりながら左詰め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 2，図 3 の下部に示す．

図の上部について，枠で囲んでいるのはページまたはキャッシュブロックを示し，1 つのブロックにちょうど 3 つのオブジェクト（ノード）が収まると仮定している．これらを見ると，2 分木のルートからノードをたどって子にアクセスするときには，幅優先ではメモリアクセスの局所性が悪くなっていることが分かる．深さ優先の場合は，左側の子にアクセスするときは局所性が良い一方，右側の子にアクセスするときは局所性が悪くなっているのが分かる．

また，図の下部を見ると，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．よって，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は，図 2，図 3 からも分かるように，幅優先順コ

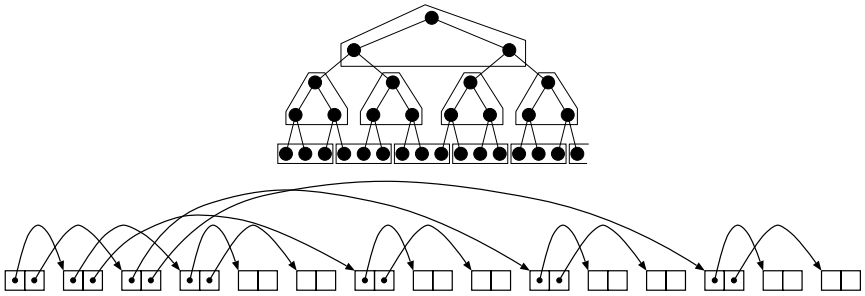

図 4 グループ化の結果
Fig. 4 Clustering copying result.

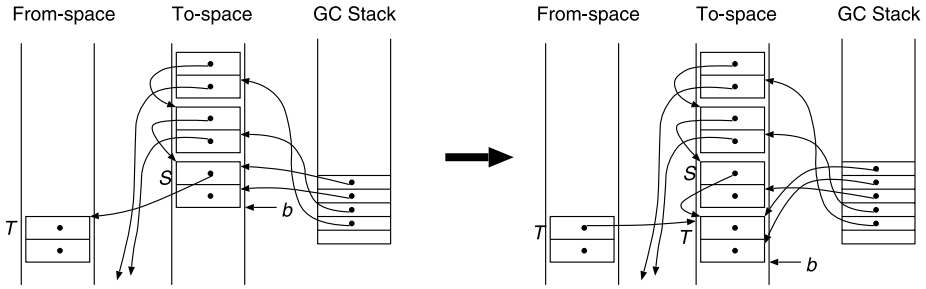

図 5 深さ優先コピー GC（S から T への参照を処理）
Fig. 5 Depth-first copying collection.

ピー GC よりも深さ優先順コピー GC の方が有利である．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良くならない．

また，高さ 5 の均一な 2 分木にグループ化を行った場合を**図 4** 上部に示す．下部は高さ 4 の均一な 2 分木についてコピー後のオブジェクトの配置である．この場合，グループ内で左右どちらの子にアクセスしても局所性が良くなっている．ただし，ページに関してグループ化した場合は，キャッシュに関しては良くなっていない可能性があり，キャッシュに関してグループ化した場合は，ページに関しては良くなっていない可能性がある．

5. 再帰的なコピー GC

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 5** に示す．From-space，To-space や，To-space の空き領域の先頭を指しているポインタ（図 5 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 To-space 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，To-space 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 To-space 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「From-space 内のオブジェクトへの参照」を To-space を向くように修正しつづけることになる（たとえば，図 5 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 1 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 5 右図のように，オブジェクト S の第 1 要素が指すオブジェクト T への参照は To-space 中の T への参照に修正される．その際，T がまだ To-space 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 5 右図では T に含まれる 2 つ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たどりたい場合は（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「From-space 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「From-space 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 To-space 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．

5.1 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 To-space 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 2 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など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，To-space 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 To-space にコピーされたオブジェクトの占める範囲全域にわたって存在するので，最新の b から To-space 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の 2 乗のオーダ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かまわないが，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 To-space 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[6] が中島らにより提案されている．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 2 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．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．

中島らは，GC スタックにおくべき要素を From-space 中のコピー済みオブジェクトが使っていたメモリを利用して記憶することで GC スタックを不要とする方式[6] も提案している．文献 4) も同様の方式である．

5.2 大部分を深さ優先順にコピーするごみ集め方式

少量のスタックで大部分を深さ優先順にコピーするごみ集め方式（限定スタック法）[7] を提案する．

基本的なアイデアは，すでに述べたように，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 Cheney の論文[1] の中で，幅

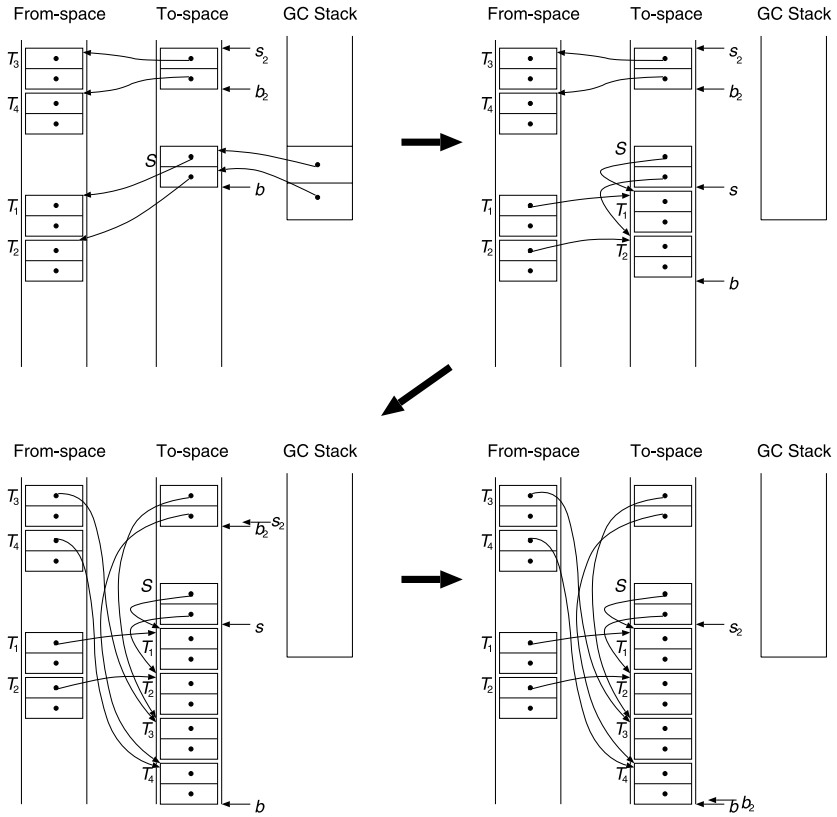

図 6 大部分を深さ優先順にコピーするごみ集め方式（限定スタック法）
（S から T_1，T_2 への参照）
Fig. 6 Mostly-depth-first copying collection with small stack space.

優先順にコピーする GC 方式を部分的に深さ優先とする方式（semi-depth-first 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，1 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 1 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 Cheney 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 semi-depth-first 方式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．From-space，To-space や，To-space 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「From-space 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，To-space 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），スタックをいったん空にする．

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）．スタックをいったん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「From-space 内のオブジェクトへの参照」を To-space 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 To-space 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 2 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 Cheney の幅優先アルゴリズムの要領で 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基にしたコピーと，s_2 と b_2 間のキューのデータを基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 間のキューをともに空にして，新しい 1 つのキューにの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基に新しい 1 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，(32 bits 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高さ 1 つ分だけ幅優先順でコピーすればよく，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．

5.3 階層的グループ化に基づくコピー GC

4 章で議論したように，深さ優先の場合は，左側の子にアクセスするときは局所性が良い一方，右側の子にアクセスするときは局所性が悪くなる．よって限定スタック法による局所性改善は限定的となる．グループ化を用いれば左右どちらの子にアクセスしてもグループ内の局所性が良くなるが，キャッシュか仮想記憶かどちらか一方についてのみしか考慮されない．

そこで，階層的にグループ化を行う方式を提案する．**図 7** に，高さ 8 の均一な 2 分木を階層的グループ化に基づく GC 方式でコピーした結果を示す．ここで，各ノードの数字はコピーされる順序である．また，ある階層レベルのグループを破線の枠で囲ってある．

単体のオブジェクトの階層レベルを 0 とする．また，階層レベル lv のグループと，そのグループから直接指されている階層レベル lv のグループをまとめて階層レベル $lv+1$ のグループを構成する．つまりレベル 0 のグループの高さは 1，レベル 1 のグループの高さは 2，レベル 2 のグループの高さは 4，レベル lv のグループの高さは 2^{lv} となる．図 7 では，最も外側の枠がレベル 3（L3 と図示）であり，その 1 つ内側がレベル 2，さらに内側がレベル 1 である．この例では，木の高さは 8 なので，最大レベルは 3 であるが，一般にはもっと大きなレベルもありうる．

提案する階層的グループ化コピー方式では，レベル lv ($lv \geq 1$) のグループ化コピーを，レベル $lv-1$ のグループ化コピーの 2 段重ねで実現する．つまり，ルートに近い側のレベル $lv-1$ のグループを（再帰的に）コピーしつつ，そのグループから外に出る参照について，その位置をレベル $lv-1$ 用の GC スタックに記録しておく．ルートに近い側のレベル $lv-1$ のグループ化コピーが完了したら，記録しておいた参照位置について（再帰的に）レベル $lv-1$ のグループ化コピーを行う．また，レベル 0 はオブジェクトであるので，従来どおりコピーする．**図 8** の `copyLV` のようにすればよい．`copyLV` のパラメータ k は，今行っているレベル lv のグループから外に出る参照について，そ

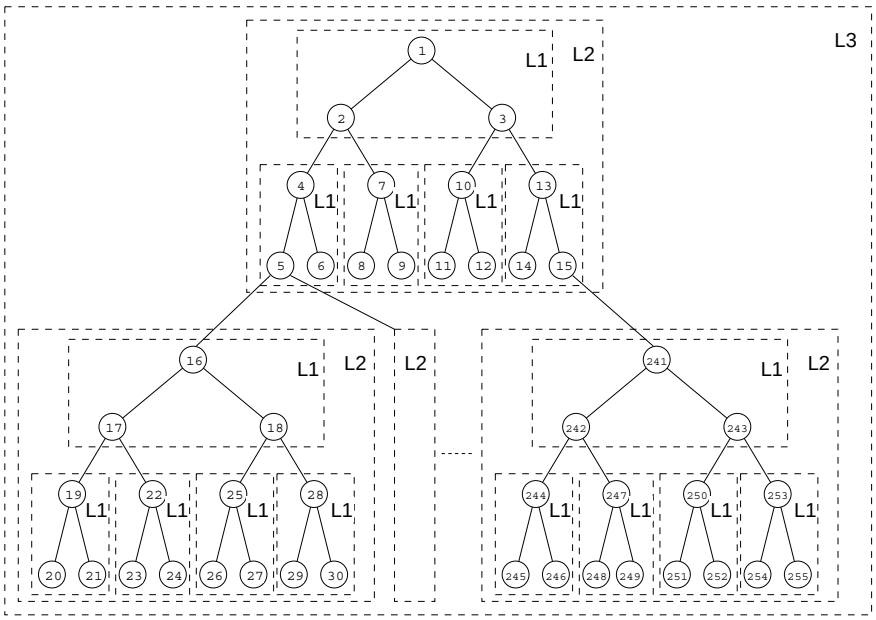

図 7 階層的グループ化の結果
Fig. 7 Hierarchical clustering copying result.

の位置をレベル k の先頭要素として集めるために指定する．

1 つのルートについて処理する場合，その処理に必要なレベルの最大値はあらかじめ分からないので，効率良く階層的グループ化コピーするにはレベルを増やしながらコピーすればよい．まとめると，1 つの GC のルートを処理する階層的グループ化コピーのアルゴリズムは図 8 のようになる．

このアルゴリズムでは，参照を保持するのにスタックを用いるので，左右の順序については図 7 のようにはならないが，本質的なことではない．左右の順序を図 7 のようにするのであればスタックの代わりに FIFO キューを用いればよいが，スタックに比べて実装が面倒になる．

この方式では，例のように 2 分木をコピーする場合，レベル n の処理では深さが 2^n までの部分木をコピーする．2 分木の深さ k のノードは最大で 2^{k-1} 個あるので，レベル n の処理では最大 2^{2^n-1} 個のオブジェクトが指す参照をスタックに保持する必要がある．このため実際の処理ではスタックあふれに注意する必要がある．スタックがあふれたときの対処には，限定スタック法と同様に幅優先キューを併用すればよい．このほかにも，あふれた参照について限定スタック法でコピーする方式[8] も考えられる．

3.2 節では仮想記憶のページや，キャッシュのブロックを意識した既存のグループ化方式[2),3),5)] について述べた．特に Chilimbi らの方式[2)] では，キャッシュのブロックサイズと連想度と容量をパラメータとして与える必要があった．提案する階層的グループ化方式では，たとえば 64 バイトといったキャッシュブロックや，たとえば 8 KB といったページのサイズをパラメータとして与えなくても，小さなレベルのグループはプロセッサのキャッシュブロックに，大きなレベルのグループは仮想記憶のページに自動的・適応的に対応しているので，キャッシュと仮想記憶の両方において,「次に」アクセスされるオブジェクトが同じブロック（ページ）に存在する確率が高いという意味で，メモリアクセスの空間的局所性を向上させることができる．これにより，キャッシュや物理メモリの容量が限られることによるキャッシュミスやページフォールトを削減できる．また，仮想記憶については TLB ミスを削減する効果も大きい．

また，木のようなデータ構造のルート付近は頻繁にアクセスされるので，つねにキャッシュ上に存在することが望ましいが，一般にはダイレクトマップ方式やセットアソシアティブ方式のように，キャッシュブロックとメモリの対応関係は限定されている（連想度が小さい）ことが多いので，2 章の最後で述べたように，頻繁にアクセスされる部分がたまたまキャッシュの同じブロックにマッピングされ，衝突によるキャッシュミスが頻発することに注意が必要である．提案する階層的グループ化方式では，木のルート付近は深さ

```
/* p に位置する参照を起点として
   レベル lv までをコピーし，含まれる
   参照の位置をレベル k の先頭要素として
   スタックに集める */
copyLV(ref *p, int lv, int k) {
  if(lv == 0) {
    従来方式と同様に*p が
    To-space を指すように変更
    その際必要ならコピーし，
    含まれる参照の位置を stack[k] へ push;
  } else {
    /* レベル lv-1 のコピーを 2 段重ねる */
    copyLV(p,lv-1,lv-1);
    while(q = pop(stack[lv-1]))
      copyLV(q,lv-1,k);
  }
}

/* p に位置する GC ルート参照を起点として
   たどれる範囲を階層的グループ化コピー
   copyLV(p, MAXLV, MAXLV) を効率良く行う */
copyHC(ref *p) {
  copyLV(p,0,0);
  for(lv=1; !isEmpty(stack[lv-1]); lv++)
    while(q = pop(stack[lv-1]))
      copyLV(q,lv-1,lv);
}
```

図 8 階層的グループ化のアルゴリズム
Fig. 8 Hierarchical clustering copying algorithm.

2^n ごとにグループ化されており，各グループ内のオブジェクトは連続したメモリに配置されている．このため「キャッシュの容量」のサイズ（たとえば 1 次キャッシュで 16 KB や 2 次キャッシュで 256 KB）以下のグループ内のオブジェクトはそれぞれ異なるキャッシュブロックに対応可能であり，頻繁にアクセスするデータについてのキャッシュ衝突ミスの問題に対応している．

6. 評　　価

評価環境として以下の構成の計算機システムを用いた．

- Sun Blade 100
  - 500 MHz UltraSPARC-IIe,
  - 64-entry iTLB,
  - 64-entry dTLB,
  - 8 KB VM pages,
  - 16 KB L1 2way set-associative 32 B-block I-Cache,
  - 16 KB L1 direct-mapped two 32 B-line (two 16-byte subblocks) D-Cache,
  - 256 KB L2 4way set-associative 64 B-line Cache,
  - 256 MB Memory,
  - Solaris 9,
  - gcc 3.2 `-O2 -mcpu=ultrasparc`
- PC
  - 1.7 GHz Pentium 4,
  - 64-entry DTLB,
  - 4 KB VM pages,
  - 12 KB L1 8way set-associative T-Cache,
  - 8 KB L1 4way set-associative 64 B-line D-Cache,
  - 256 KB L2 8way set-associative 128 B-line (two 64 B sectors) Cache,
  - 512 MB Memory,
  - Linux 2.4.20,
  - gcc 3.2 `-O2`

また，以下の評価において，GC スタックのサイズはスタックあふれの処理が事実上起こらないように調整し，1 スタックあたり 256 K ワードに設定した．

6.1 2 分探索木の場合

各計算機システムで 2 分探索木のプログラムを実行して実行時間を計測した結果を**図 9**，**図 10** に示す．ランダムな key のデータをある回数挿入し 2 分探索木を構成した後，明示的に GC を起動し，その後ランダムな key で探索を繰り返し行う．図 9，図 10 には，2 分探索木に挿入済みのオブジェクト数を横軸として 1 回の探索あたりの平均の探索時間を示している．ただし，ランダム key の生成コストも含んでいる．横軸は log スケールとしているが，2 分探索木の平均的高さや探索操作でトラバースする平均的深さに比例するといえる．比較するコピー方式の種類には，GC なし（no-GC），Cheney の幅優先（BF），Moon の方式に基づく 4 KB 単位のグループ化（CLp），限定スタック法の深さ優先（DF），階層的グループ化方式（HC），レベル 1 の階層のみグループ化しグループは深さ優先とした方式（CL1-DF）を用いた．CL1-DF は，HC の実装を少し修正することで得ることができ，スタックあふれへの対応などは HC の実装と同様になる．2 分探索木のノードオブジェクトのサイズはヘッダを含めて 20 B であり，8 B 境界で割り当てるため各オブジェクトが 24 B 消費する．たとえば，32×10^5 個のオブジェ

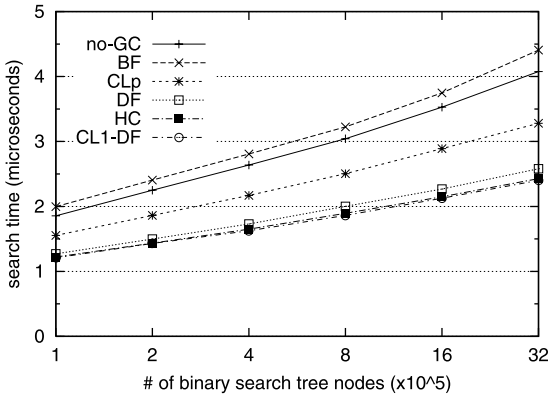

図 9 Pentium 4 における 2 分探索木の探索時間
Fig. 9 Search time for binary search tree on Pentium 4.

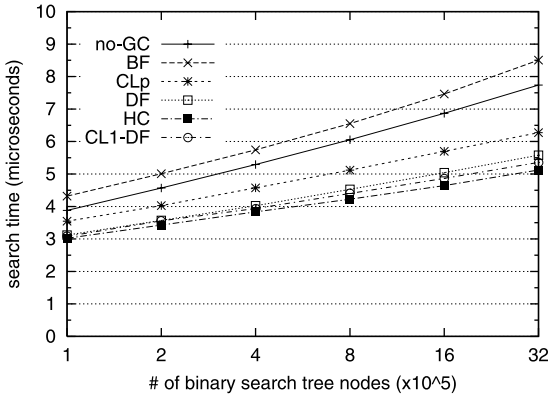

図 10 UltraSPARC-IIe における 2 分探索木の探索時間
Fig. 10 Search time for binary search tree on UltraSPARC-IIe.

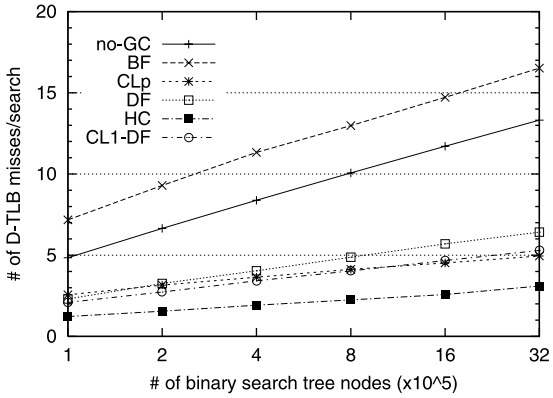

図 11 UltraSPARC-IIe における 2 分探索木の探索時の TLB ミス回数
Fig. 11 TLB-misses for binary tree search on UltraSPARC-IIe.

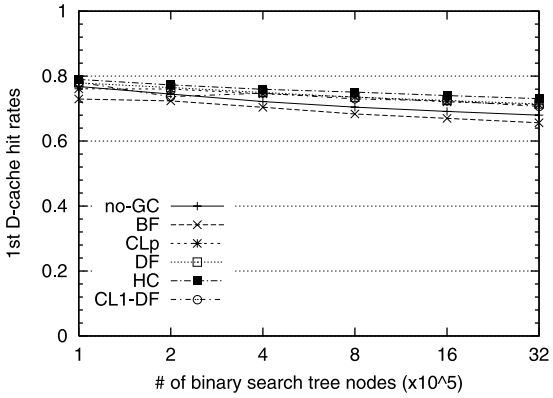

図 12 UltraSPARC-IIe における 2 分探索木の探索時の 1 次データキャッシュ（read）ヒット率
Fig. 12 D-cache (read) hit rates for binary tree search on UltraSPARC-IIe.

クトで，約 73.2 MB のメモリを使用する．このサイズのときの各種方式の性能を比較すると，BF は no-GC と比較して Pentium 4 ベースのシステムでは 7.5%，UltraSPARC-IIe ベースのシステムでは 9.0%低下している．また，CLp は BF と比較して 34%（Pentium 4）または 36%（UltraSPARC-IIe）向上している．また，DF は BF と比較して 71%（Pentium 4）または 52%（UltraSPARC-IIe）向上している．また，HC は BF と比較して 82%（Pentium 4）または 66%（UltraSPARC-IIe）向上している．階層的グループ化により幅優先と比較して Pentium 4 の場合に 1.82 倍もの性能向上が得られたのは 1 つには Pentium 4 の 2 次キャッシュの（読み込み時の）ブロックサイズが 128 B と大きくグループ化の効果が大きいためと考えられる．

また，CL1-DF は Pentium 4 の場合には HC よりわずかに高い性能（図 9 ではほぼ重なっている），UltraSPARC-IIe の場合（図 10）には HC に及ばない性能であった．

UltraSPARC-IIe ベースの計算機システムについては，Solaris 9 が提供する測定ユーティリティを用いて，探索 1 回あたりのデータ TLB ミス回数（**図 11**），1 次データキャッシュの読み込み時ヒット率（**図 12**），2 次キャッシュの読み込み時ヒット率（**図 13**）を測定した．データ TLB ミスについては，GC 方式間の差が顕著に現れている．階層的グループ化（HC）の効果が高いことが分かる．1 次・2 次キャッシュヒット率についても HC が良い．また，2 次キャッシュについてはレベル 1 のみグループ化も同等の性能を示した．ここで，CLp は TLB ミスについては DF より良いが，2 次キャッシュミスについては DF より悪く，総合的には DF より遅くなっている．

以上の性質は，**図 14** に示す，参照をたどるようなアクセスにおける参照元–参照先オブジェクト間距離

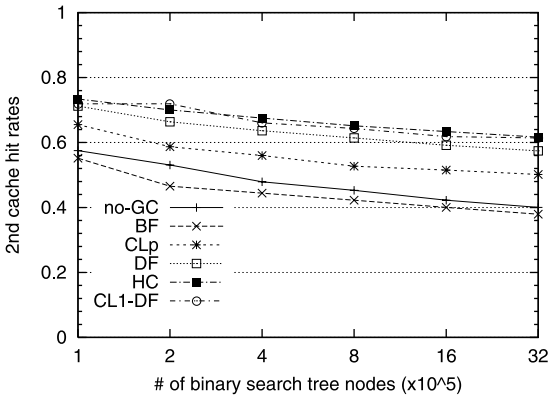

図 13 UltraSPARC-IIe における 2 分探索木の探索時の 2 次キャッシュ（read）ヒット率
Fig. 13 2nd-cache (read) hit rates for binary tree search on UltraSPARC-IIe.

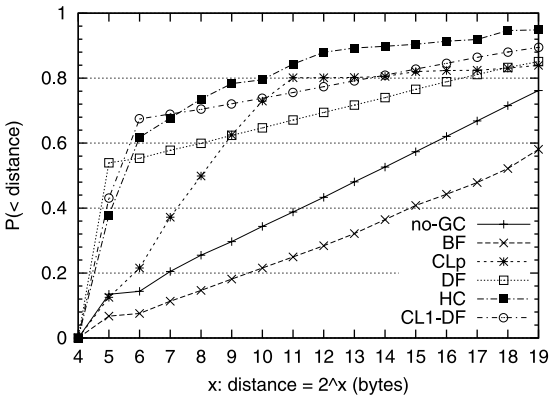

図 14 32×10^5 要素上の 2 分探索における参照元–参照先間距離の累積分布
Fig. 14 Cumulative distribution of referrer-referee distances in binary search on 32×10^5 elements.

の累積分布からも説明できる．

6.2 表と連想リストの場合

前節と同様の条件で，表と連想リストを用いる探索プログラムを実行して実行時間を計測した結果を**図 15**，**図 16** に示す．表は 16 エントリの小表を 4 段用いて 65536 エントリとした．連想リストは car 部と cdr 部からなるセル（16 B を消費）を用いて，car 部に key と value のペア（16 B を消費）へのポインタ，cdr 部に続きの連想リストを持つようにした．連想リストは key について昇順となるように登録し，途中で探索が打ち切れるようにしている．

1×10^5 個の要素を記憶させた場合の性能を比較してみると，小表部分の寄与が大きいので HC が一番良い性能を示している．BF は no-GC より Pentium 4 ベースのシステムでは 6.9%，UltraSPARC-IIe ベースのシステムでは 5.7%低下している．CLp の性能は，BF と比較して 12%（Pentium 4）または 7.6%（UltraSPARC-IIe）向上している．DF の性能は，BF と比較して 31%（Pentium 4）または 15%（UltraSPARC-IIe）向上している．HC の性能は BF と比較して 38%（Pentium 4）または 21%（UltraSPARC-IIe）向上している．CL1-DF の性能は BF と比較して 37%（Pentium 4）または 17%（UltraSPARC-IIe）向上している．

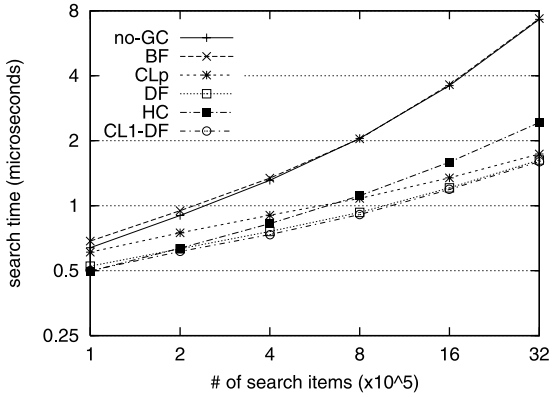

図 15 Pentium 4 における表+連想リストの探索時間
Fig. 15 Search time for table+associative list on Pentium 4.

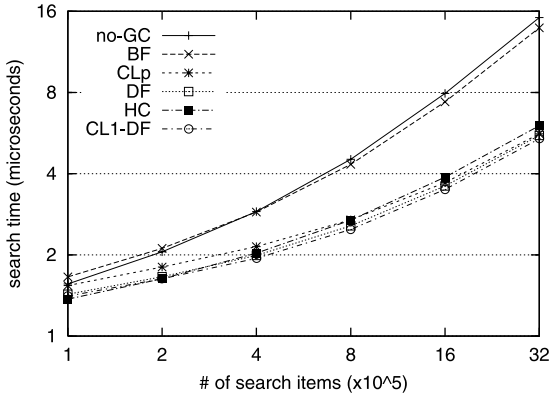

図 16 UltraSPARC-IIe における表+連想リストの探索時間
Fig. 16 Search time for table+associative list on UltraSPARC-IIe.

32×10^5 個の要素を記憶させた場合の性能を比較してみると，リスト部分が長くなりデータ構造の形状がレベル 1 のみグループ化（CL1-DF）に有利になっているため，これが一番良い性能を示している．BF は no-GC よりすこしだけ性能が良い．CLp の性能は，BF と比較して 4.24 倍（Pentium 4）または 2.47 倍（UltraSPARC-IIe）向上している．DF の性能は，BF と比較して 4.52 倍（Pentium 4）または 2.50 倍（UltraSPARC-IIe）向上している．HC の性

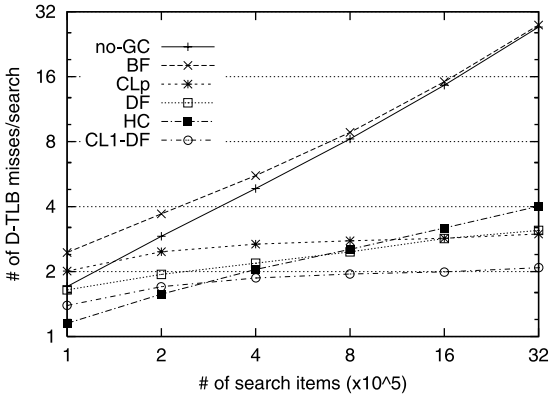

図 17 UltraSPARC-IIe における表+連想リストの探索時の TLB ミス回数

Fig. 17 TLB-misses for table+associative list search on UltraSPARC-IIe.

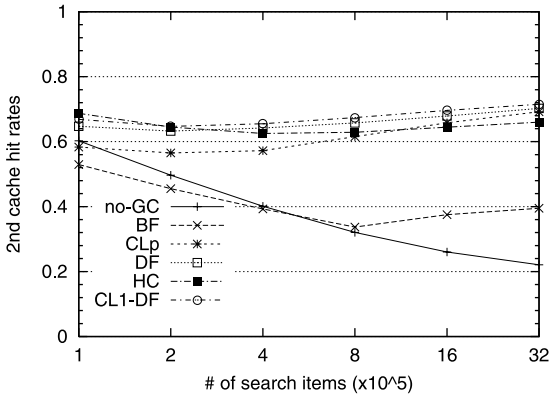

図 19 UltraSPARC-IIe における表+連想リストの探索時の 2 次キャッシュ（read）ヒット率

Fig. 19 2nd-cache (read) hit rates for table+associative list search on UltraSPARC-IIe.

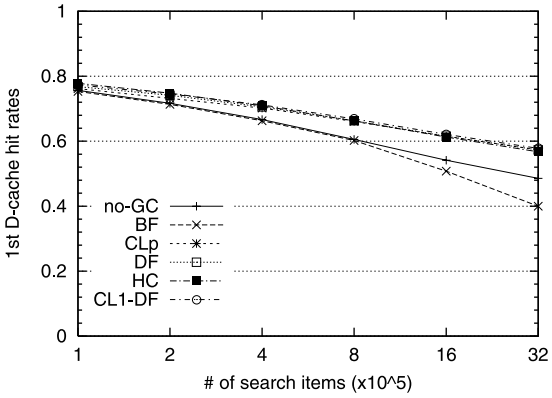

図 18 UltraSPARC-IIe における表+連想リストの探索時の 1 次データキャッシュ（read）ヒット率

Fig. 18 D-cache (read) hit rates for table+associative list search on UltraSPARC-IIe.

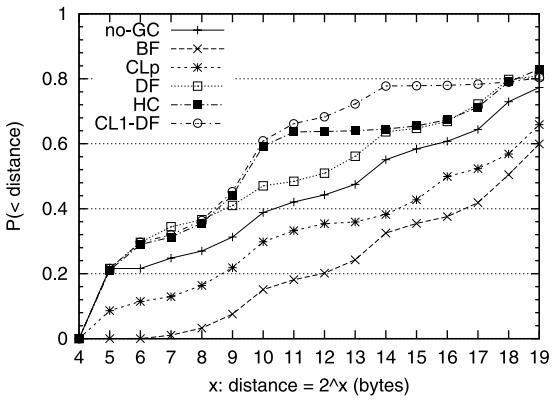

図 20 10^5 要素上の表+連想リスト探索における参照元–参照先間距離の累積分布

Fig. 20 Cumulative distribution of referrer-referee distances in table+associative list search on 10^5 elements.

能は BF と比較して 3.03 倍（Pentium 4）または 2.31 倍（UltraSPARC-IIe）向上している．CL1-DF の性能は BF と比較して 4.60 倍（Pentium 4）または 2.58 倍（UltraSPARC-IIe）向上している．

UltraSPARC-IIe ベースの計算機システムについては，探索 1 回あたりのデータ TLB ミス回数（**図 17**），1 次データキャッシュの読み込み時ヒット率（**図 18**），2 次キャッシュの読み込み時ヒット率（**図 19**）を測定した．データ TLB ミスについては，GC 方式間の差が顕著に現れている．要素数が少ないとき階層的グループ化（HC），レベル 1 のみグループ化（CL1-DF）が TLB ミスが少ない．要素数が増えるに従い，BF は極端に TLB ミスが増え，HC，DF も，CLp，CL1-DF と比較して TLB ミスが増える．HC で TLB ミスが増えるのは連想リストの途中にグループ化の切れ目がきてしまうためである．リストのように広がりを持たない場合はグループの高さを大きくするなどの工夫が今後必要であると考えられる．1 次データキャッシュについていえば，ヒット率は要素数が増えるに従い no-GC と BF が悪くなることが分かる．また，2 次キャッシュについては TLB ミスと似た傾向にあることが分かる．

以上の性質は，**図 20**，**図 21** に示す，参照をたどるようなアクセスにおける参照元–参照先オブジェクト間距離の累積分布からも説明できる．

6.3 スタックサイズ

スタックがあふれた場合，高さ 1 つ分だけ幅優先順でコピーするが，探索においては探索ごとにこの「段差」を横切ることが多く，性能への影響は比較的大きい．このためスタックあふれはできるだけ発生しないようにするのがよい．

前節までの評価では GC スタックのサイズは，ス

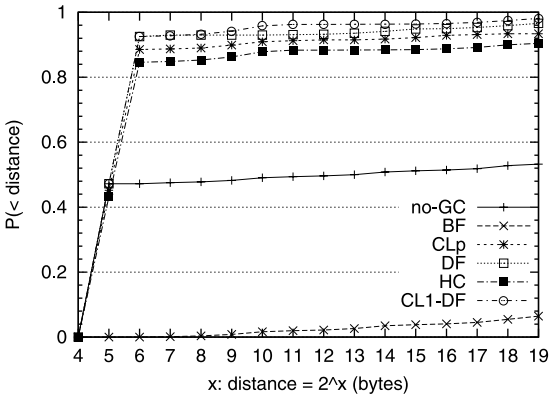

図 21 32×10^5 要素上の表+連想リスト探索における参照元–参照先間距離の累積分布

Fig. 21 Cumulative distribution of referrer-referee distances in table+associative list search on 32×10^5 elements.

タックあふれの処理がほとんど起こらないように 1 スタックあたり 256 K ワード（1 MB（$= 2^{20}$B））とした．評価では 32×10^5 要素の 2 分探索木で階層的グループ化を用いた場合のみ，GC 中に 1 回のスタックあふれが発生した．このときの生きているオブジェクトの総量は約 73.2 MB である．コピー GC には From-space，To-space 合わせて約 146.5 MB のヒープが必要なことを考えると，1 MB というのは 1%以下のサイズである．限定スタック法の場合はこれで十分すぎるくらいで，実際には 1 K ワードもあれば十分である．

階層的グループ化の場合，階層ごとに 1 つ GC スタックを用いており，32 ビットアドレス空間なら理論上は 32 程度の GC スタックが必要である．ただし，評価でのスタックあふれは階層レベル 5（高さ 32）用のスタックで生じていることから考えると，あまり大きなレベルの階層は用いないようにしたり，階層間でスタック用の領域を融通したりすることで GC スタックの総量は無視できるサイズにできると考えられる．

7. おわりに

本研究では，まず，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ごみ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ればメモリアクセスの局所性を改善することができる．本研究では，さらにメモリアクセスの局所性を改善する，オブジェクトを階層的にグループ化してコピーする方式を提案した．また，評価結果により提案方式の有効性を示した．特に，階層的グループ化は，特定のスケールのみの局所性を高める関連研究[2),3),5)] と違い，キャッシュブロックやページなどの様々なスケールにおいて局所性を高める効果を持つ．

実際のコピー GC では，世代別 GC などが用いられるが，本研究のようなオブジェクトのコピー順は，主に寿命の長い旧世代空間へ適用するとよい．

今後は，他の様々なデータ構造やプログラムについても測定をすると興味深い結果が得られると思われる．また提案するごみ集め方式の並列化も検討している．並列計算機では優れた負荷分散が重要となるが，提案する方式はスタックを利用してい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．

謝辞 本研究の一部は，文部科学省科学研究費特定（2）13324050 の補助を得て行った．

参考文献

1) Cheney, C.J.: A Nonrecursive List Compacting Algorithm, *Comm. ACM*, Vol.13, No.11, pp.677–678 (1970).
2) Chilimbi, T.M., Hill, M.D. and Larus, J.R.: Cache-Conscious Structure Layout, *Proc. PLDI'99*, pp.1–12 (1999).
3) Moon, D.A.: Garbage Collection in a Large Lisp System, *Proc. Conference on Lisp and Functional Programming*, pp.235–246 (1984).
4) Thomas, S.P.: Garbage collection in shared-environment closure reducers: Space-efficient depth first copying using a tailored approach, *Information Processing Letters*, Vol.56, No.1, pp.1–7 (1995).
5) Wilson, P.R., Lam, M.S. and Moher, T.G.: Effective "Static-graph" Reorganization to Improve Locality in Garbage-Collected Systems, *ACM SIGPLAN Notices*, Vol.26, No.6 (*Proc. PLDI'91*), pp.177–191 (1991).
6) 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 (1995).
7) 八杉昌宏，伊藤智一，小宮常康，湯淺太一：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，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).
8) 伊藤智一，八杉昌宏，小宮常康，湯淺太一：局所性を高める階層的コピー GC 方式，日本ソフトウェア科学会第 19 回大会論文集，No.5A–3 (2002).

(平成 15 年 9 月 22 日受付)
(平成 15 年 11 月 14 日採録)

八杉　昌宏（正会員）

1967 年生．1989 年東京大学工学部電子工学科卒業．1991 年同大学大学院電気工学専攻修士課程修了．1994 年同大学院理学系研究科情報科学専攻博士課程修了．1993 年〜1995 年日本学術振興会特別研究員（東京大学，マンチェスター大学）．1995 年神戸大学工学部助手．1998 年京都大学大学院情報学研究科通信情報システム専攻講師．2003 年より同大学助教授．博士（理学）．1998 年〜2001 年科学技術振興事業団さきがけ研究 21 研究員．並列処理，言語処理系等に興味を持つ．日本ソフトウェア科学会，ACM 会員．

伊藤　智一

1977 年生．2001 年京都大学工学部情報学科卒業．同年より同大学大学院情報学研究科修士課程に在学中．言語処理系等に興味を持つ．

小宮　常康（正会員）

1969 年生．1991 年豊橋技術科学大学工学部情報工学課程卒業．1993 年同大学大学院工学研究科情報工学専攻修士課程修了．1996 年同大学院工学研究科システム情報工学専攻博士課程修了．同年京都大学大学院工学研究科情報工学専攻助手．1998 年同大学院情報学研究科通信情報システム専攻助手．2003 年より豊橋技術科学大学情報工学系講師．博士（工学）．記号処理言語と並列プログラミング言語に興味を持つ．平成 8 年度情報処理学会論文賞受賞．

湯淺　太一（フェロー）

1952 年神戸生．1977 年京都大学理学部卒業．1982 年同大学大学院理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987 年豊橋技術科学大学講師．1988 年同大学助教授，1995 年同大学教授，1996 年京都大学大学院工学研究科情報工学専攻教授．1998 年同大学院情報学研究科通信情報システム専攻教授となり現在に至る．理学博士．記号処理，プログラミング言語処理系，並列処理に興味を持っている．著書「Common Lisp 入門（共著）」,「C 言語によるプログラミング入門」,「コンパイラ」ほか．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．