2008-4-(7): 情報処理学会プログラミング研究会 発表資料 2009 年 1 月 27 日

階層的グループ化コピーアルゴリズムにおける Cache-oblivious 配置法と Cache-conscious 配置法の実験的比較

八杉昌宏[†1] 後藤裕輔[†1]
馬谷誠二[†1] 湯淺太一[†1]

プロセッサとメモリの性能差は拡大していることから，仮想記憶の局所性とともにキャッシュの局所性の改善は重要な課題である．ポインタベースの大規模データ構造上の探索などでは，データオブジェクトの配置が悪いと，キャッシュミス，TLB ミス（場合によっては，ページフォールト）が多発する．このため，コピー型ごみ集めアルゴリズムにおいては，幅優先順，深さ優先順以外に，グループ化された配置を実現するアルゴリズムも提案されてきた．本研究では「階層的グループ化」を提案している．これは，データオブジェクトを複数の階層レベルにおいてグループ化するものであり，メモリ階層上のキャッシュレベル，仮想記憶レベルの双方において局所性を改善する．例えば，2 分探索木上の探索，および連想リストの配列上の探索において，従来よく用いられてきた幅優先のコピーアルゴリズムと比較して，約 2〜5 倍の性能向上が得られた．本論文では以前提案した複数のスタックを用いた cache-oblivious 配置アルゴリズムを発展させた，階層レベル毎のスキャンポインタを使う cache-conscious 配置アルゴリズムを提案し，両者の実験的比較について報告する．

An Experimental Comparison of Cache-oblivious and Cache-conscious Techniques in Hierarchical-clustering Copying Algorithms

Masahiro Yasugi,[†1] Yusuke Goto,[†1] Seiji Umatani[†1]
and Taiichi Yuasa[†1]

The increasing processor-memory performance gap makes improving the cache locality as important as the virtual memory locality. In some applications, such as search algorithms on pointer-based large data structures, locality-poor data-object placement significantly increases cache misses and TLB misses (and page faults in some cases). Thus, for garbage collection, several clustering copying algorithms have been proposed instead of breadth-first copying and depth-first copying. In this research, we have proposed "hierarchical clustering" which groups data objects at multiple hierarchical levels and provides better locality at both the cache and virtual memory levels of the memory hierarchy. For example, when employing a binary search tree and an array of associative lists, our algorithms achieved approximately two (sometimes five) times speedups compared to conventional breadth-first copying algorithms. In this paper, we propose a new copying algorithm that achieves cache-conscious data placement with multiple scan pointers for their own levels of memory hierarchy, developed from our previously proposed algorithm that achieves cache-oblivious data placement with multiple stacks; we report an experimental comparison of these two algorithms.

1. はじめに

近年，プロセッサとメモリとの性能差は拡大しており，高性能を達成するには仮想記憶の局所性とキャッシュの局所性の改善はともに重要な課題である．キャッシュについては，その階層（L1 キャッシュ，L2 キャッシュ等），容量，ラインサイズ，連想度（way 数）などを意識してキャッシュミスを削減した cache-conscious なプログラムや，個々の特徴は意識しないがメモリアクセスに様々なスケールの局所性を持たせて様々なキャッシュに自動的・適応的に対応する cache-oblivious なプログラムが考えられる．仮想記憶については，十分な主記憶によりページフォルトが削減できたとしても，TLB（Translation Lookaside Buffer）ミスの削減も重要であり，例えば，UltraSPARC ベースのプロセッサでは TLB ミスはトラップとしてソフトウェアで処理される．本論文では，仮想記憶に関して，主記憶容量，ページサイズや境界，TLB の特徴（エントリ数，連想

†1 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University

度，階層）などを意識してページフォールトや TLB ミスを削減する場合についても便宜上 cache-conscious と呼ぶことにする[⋆1]．

局所性を改善するには，プログラム上のメモリアクセスの順序を工夫する方法と，データ配置を工夫する方法があるが，本研究では特にポインタで指されるデータオブジェクトの配置法を扱う．自動的なメモリ領域管理（ごみ集め，garbage collection，GC と略す）をサポートするプログラミング言語の実装には，言語のセマンティクスを保存する限りにおいてヒープ中のデータオブジェクトの配置を再構成する自由とともに，局所性改善の機会が与えられている．

多くのアプリケーション（特にポインタベースの大規模なデータ構造上の探索など）において，データオブジェクトを**幅優先**順にコピーしてしまうアルゴリズム（標準的な Cheney の非再帰コピー型 GC アルゴリズム[4]）を用いると，キャッシュミス，TLB ミス，ページフォールトが増加する．**深さ優先順**のコピーアルゴリズム[14),19),20)] により局所性は改善されるが，改善は限定的であり，**グループ化**に基づくコピーアルゴリズム[10),16)] も提案されてきた．

本研究では「**階層的グループ化**」を提案している[17),20)]．これは，**図 7** のように，データオブジェクトのグループ（階層レベル 1，L1 と略す），L1 のグループのグループ（L2）…のように，データオブジェクトを複数の階層レベルにおいてグループ化するものであり，メモリ階層上のキャッシュと仮想記憶の双方において優れた局所性をもたらす．提案手法は特に，局所性を持たなければ性能低下が顕著な，ポインタベースの大規模なデータ構造上の探索などで，キャッシュミスと TLB ミスを**同時に**削減するのに有効である．

本論文では，階層的グループ化のための新しい cache-conscious なコピーアルゴリズム[17)] を提案する．このアルゴリズムは以前提案した，ある程度の作業領域を必要とする cache-oblivious なアルゴリズム[20)] と異なり，後に**図 8** で示すような階層レベル毎のスキャンポインタを使うほかにはほとんど作業領域を必要としない．各スキャンポインタが優先度を持ち，それぞれが担当するメモリ階層の局所性を改善する．本コピーアルゴリズムは大規模データ構造上の探索のほとんどすべての場合において既存アルゴリズムよりよい性能を得た．実験では，(1) 2 分探索木上の探索，(2) 2 分探索木配列上の探索，および (3)（ある種のオープンハッシュ表を模擬した）連想リスト配列上の探索において，従来よく用いられてきた幅優先順コピーアルゴリズムと比較して，約 2〜5 倍の性能向上が得られた．

本論文の主要な貢献は，cache-conscious 配置法の提案と，提案手法と既存手法との実験上の比較である．

以下，2 章ではコピー型 GC と局所性について議論する．3 章では，既存の「グループ化」アルゴリズムについて復習する．4 章では我々が提案している「階層的グループ化」について紹介するとともに，新しい cache-conscious なコピーアルゴリズム[17)] を提案する．5 章では我々がすでに発表している cache-oblivious なアルゴリズム[20)] とスタック溢れ対策について簡単に復習する．6 章は実装について述べ，7 章で性能評価を行い，8 章で関連研究について述べる．

⋆1 主にページサイズ・ページ境界を意識する．

```
pointer free;       // the free pointer
pointer scan;       // the scan pointer

/* copy live objects referred to by
   pointers between scan and free. */
scavenge()
{
  while (scan < free)
   {
    foreach (link in CHILDREN(scan))
      if (FROM_SPACE(*link))
        if (TO_SPACE(FWD_PTR(*link)))
          *link = FWD_PTR(*link);
        else
         {
           MOVE(*link, free);
           FWD_PTR(*link) = free;
           *link = free;
           free += SIZE(free);
         }
     scan += SIZE(scan);
   }
}
```

図 1 Cheney の非再帰的コピーアルゴリズム
Fig. 1 Cheney's nonrecursive copying algorithm.

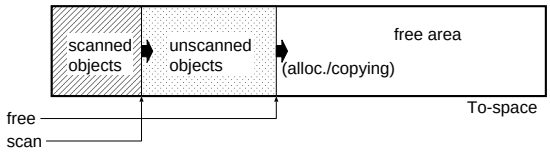


図 2 Cheney のアルゴリズムにおける To 空間
Fig. 2 To-space of Cheney's algorithm.

2. コピー型ごみ集めと局所性

コピー GC は 2 つの半空間（それぞれ From 空間と To 空間と呼ぶ）の間で，すべての生きているオブジェクトを移動させることで，不要なオブジェクトを

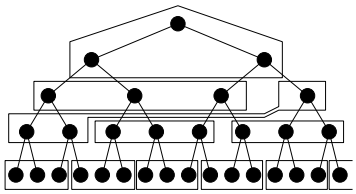


図 3 幅優先順コピーの結果
Fig. 3 Breadth-first copying result.

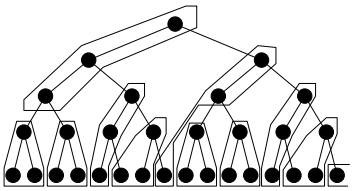


図 4 深さ優先順コピーの結果
Fig. 4 Depth-first copying result.

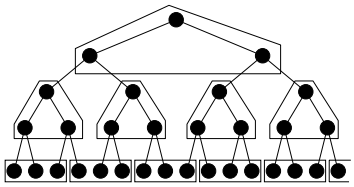


図 5 グループ化コピーの結果
Fig. 5 Clustering copying result.

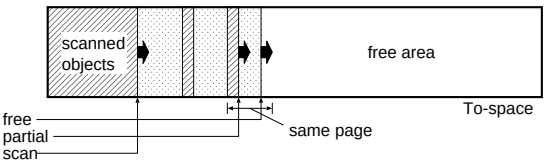


図 6 Moon のアルゴリズムにおける To 空間
Fig. 6 To-space of Moon's algorithm.

除去する方式である．ここでのコピー順は，ごみ集めの正しさには無関係だが，コピーコレクタ（copying collector，コピー型ごみ集め器）自体の局所性にとっては重要であり，それにもましてコピー後のオブジェクトを使用するミューテータ（mutator，プログラムの通常の実行を進める主体）の局所性にとって重要である．

Cheney の非再帰コピーアルゴリズム[4] は To 空間中の動的な区間を未スキャンのオブジェクト（フィールド）のキューとして用いる．このため，スキャンポインタとフリー（free）ポインタ以外の余分な作業空間（スタック等）を特に必要としない．そのアルゴリズムを**図 1** および**図 2** に示す．生きているオブジェクトそれぞれは To 空間中の（`free` ポインタが指す）次の free なメモリ領域に余計な隙間なく連続的にコピーされる．コピーされたオブジェクトはいずれ（`scan` ポインタにより）スキャンされ，新しくコピーすべき生きたオブジェクトへの参照を見つけるか，(共有データ構造などで）すでにコピーされた転送ポインタ（`FWD_PTR`）を持つオブジェクトの場合はコピー先に参照を修正する．

しかし，このアルゴリズムではオブジェクトが幅優先順にコピーされてしまう．多くのアプリケーションで，幅優先順コピーは空間的局所性を悪化させ，キャッシュミス，ページフォールト，TLB ミスを増加させる．**図 3** では，同一のキャッシュブロック（あるいはページ）に含まれるオブジェクトを表しているが，ルート付近を除いて，あまり関係のないオブジェクトらがグループ化されてしまうことが分かる．

深さ優先順の配置は一般に幅優先順よりもよい局所性をもたらすが，スタックなどの余分な作業空間を必要とする．さらに，深さ優先順配置による局所性改善は限定的である．つまり，**図 4** に示すように，左側のリンクを辿るときには，同じキャッシュブロック（あるいはページ）に関する再利用がうまくなされるが，右側のリンクを辿った場合は，末端オブジェクト付近を除いては別のキャッシュブロック（あるいはページ）に到達してしまい，局所性は良くない．

3. グループ化

深さ優先順コピーによる局所性改善は限定的であるため，いくつかの「グループ化（clustering)」コピーアルゴリズムが提案されてきた[10),16)]．

Moon は，近似的深さ優先（approximately depth-first）アルゴリズム[10)] を提案した．これは，当初深さ優先を意図していたが，実際のところはグループ化アルゴリズムとなっている．互いに関連するオブジェクトは，同じ仮想記憶のページに，**図 5** のようにグループ化される．その実現のために第 2 のスキャンポインタを導入する．このポインタは**図 6** において `partial` と示されており，途中まで埋められたページの中を指している．つまり，この第 2 のスキャンポインタと free ポインタはいつも同じページを指すようにする．両者の間の未スキャンの領域は，第 1 のスキャンポインタがスキャンするより「先に」第 2 のスキャンポインタによりスキャンされることになる．このように To 空間のある割合のオブジェクトは，2 つのスキャンポインタにより重複してスキャンされることになる．Moon はこの方式をページレベルに適用しているが，我々は Moon の方式がキャッシュレベルでのグループ化にも適用可能であることを指摘しておきたい．

Wilson らは，hierarchical decomposition[16)] を提

案している．これは各ページにそれ専用のスキャンポインタと free ポインタのペアを準備し，Moon のアルゴリズムにおける再スキャンを消去したものとなっている．しかし，この Wilson-Lam-Moher のアルゴリズムは，直接キャッシュレベルに適用するには，準備する scan/free ポインタのペアが多すぎるという問題がある．なお，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 2 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改良でも，これらの利点を取り入れていく必要がある．

Chilimbi らは cache-conscious なデータ配置[5] を提案している．彼らの手法はごみ集めシステムに関してではないが，キャッシュレベルの「グループ化」を行っており，すぐれた性能を得ている．残念ながら彼らのアルゴリズムの詳細は不明であるが，おそらくある種の再帰的アルゴリズムを用いていると予想される．この点は Moon のアルゴリズムと異なることになる．

これらの「グループ化」コピーアルゴリズムはメモリ階層の特定の側面にのみ着目している．例えば，Moon のアルゴリズムと Wilson-Lam-Moher のアルゴリズムは仮想記憶のページにあわせたグループ化を行う．このとき，ページ内のオブジェクトの順序は幅優先順となり，キャッシュレベルの局所性は乏しくなってしまう．逆に本来の意図に反して Moon のアルゴリズムと Wilson-Lam-Moher のアルゴリズムをキャッシュレベルに適用することも考えられるが，キャッシュにあわせたグループ群が，Moon のアルゴリズムでは semi-depth-first 順[4] に，Wilson-Lam-Moher のアルゴリズムでは幅優先順にコピーされ，仮想記憶レベルの局所性には問題を残す．

4. 提 案 方 式

以下，4.1 節では我々が提案している「階層的グループ化（hierarchical clustering)」という再配置の形について説明する．これは，メモリ階層におけるキャッシュと仮想記憶の双方において優れた局所性をもたらす．

4.3 節では，複数のスキャンポインタと限られた作業空間でキャッシュを意識した「階層的グループ化」を行うコピー GC アルゴリズムを提案する．4.1 節の「階層的グループ化」の定義では，共有データ構造や循環データ構造に関しては定まっているとはいえないが，提案するアルゴリズムはこの曖昧性を取り除くことになる．具体的には Cheney のアルゴリズムと同様に転送ポインタを用いてコピー済オブジェクトへの参照を修正する．

4.1 階層的グループ化

「階層的グループ化」の例は図 7 のように示せる．この例では，高さ 8 の均質な 2 分木を用いており，3 つの階層レベル （図の `L1`，`L2`，`L3`）を用いる．図では破線で各階層レベルのグループを表している．

各オブジェクトを，コピー後のオブジェクトの位置を示す番号付の円で表す．ある（任意の階層レベルの）グループのすべてのオブジェクトは番号順に連続的に配置される．(例えば “1 から 3”， “4 から 6”，“16 から 30” のようになる．)

我々の再帰的な「階層的グループ化」の定義は次のようになる．階層レベル l ($l = 1, 2, 3, \ldots$) の各グループは，階層レベル $l-1$ の先頭グループとそれに続く 0 個またはそれ以上個の連続的に配置された階層レベル $l-1$ の派生グループ（ら）から構成される．ここで，各オブジェクトは階層レベル 0 の単一集合グループとみなす．階層レベル l ($l = 0, 1, 2, 3, \ldots$) の各グループは，単一の先頭オブジェクトを持ち，それは自分自身であるか ($l = 0$) または階層レベル $l-1$ の先頭グループの先頭オブジェクトである ($l = 1, 2, 3, \ldots$)．C をレベル l ($l = 1, 2, 3, \ldots$) のグループとすると，C 中の任意の派生グループ（レベル $l-1$）の先頭オブジェクトは，C の先頭グループ（レベル $l-1$）から直接参照されているか，C 中の他の派生グループ（レベル $l-1$）を介して間接的に参照されている．ここで，C を構成する階層レベル $l-1$ のグループ間を結ぶポインタの参照元・参照先はともに C に含まれる．もちろん，C 内部の階層レベル $l-1$ のグループ内部のポインタの参照元・参照先も含むことになる．

4.2 Cache-conscious 階層的グループ化

図 7 は cache-oblivious 配置として見る方が適しているが，これについては次章で述べる．キッシュを意識した階層的グループ化（cache-conscious hierarchical clustering，CC-HC と略す）配置とするには，図 7 の

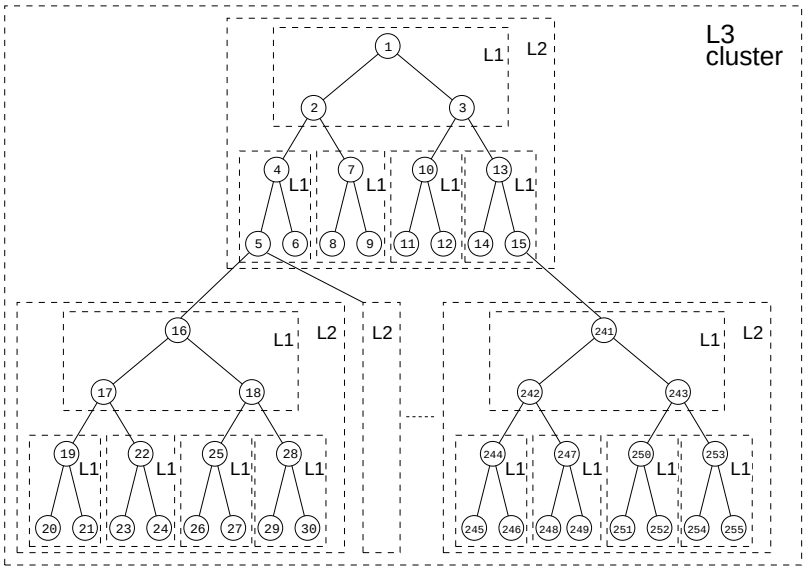


図 7 階層的グループ化の例（cache-oblivious の例でも，cache-conscious の特殊な例でもある.）
Fig. 7 An example of hierarchical clustering.

L1，L2 といった各階層レベルをそれぞれメモリ階層の 1 側面に対応させる．このため，例えば図 7 では L1 グループは 3 つのオブジェクトからなる部分木としたが，CC-HC 配置の場合は，そのために適切な（キャッシュブロックの境界等を考慮した）個数のオブジェクトからなる部分木とする．

我々のアルゴリズムは，各レベルのグループを対応するメモリ階層の側面の望ましい条件に合わせようとする．このとき，あるグループの最後のオブジェクトは望ましいサイズより大きかったり，望ましい境界を超えなくては配置できなかったりもするため，条件は正確に満たされるとは限らない．これは先頭オブジェクトもまた，望ましい境界の切りのよいところから配置できるという保証もないことを意味する.（もちろん，GC でなければ適当に隙間を許すことも考えられる.）

典型的なキャッシュと仮想記憶の局所性改善のためには，階層レベル L1 のグループを L2 キャッシュのブロック（転送単位）に合わせ，階層レベル L2 のグループを仮想記憶のページに合わせることを推奨する．前者にはキャッシュの義務的ミスや容量ミスを削減する効果があり，後者には TLB ミスやページフォールトを削減する効果がある．最後に階層レベル L3 を To 空間全体とすることで，図 8 に示すように，全体のコピー GC を進めることができる．この場合，最大の階層レベルは 3 ということになる.（つまり階層レベル 0 から 3 が使われる.）

特に連想度の低いキャッシュではコンフリクトミス削減のための工夫も重要となる．特に探索木における木のルート付近は，どのような探索でも共通してアクセスされる重要な部分であり，そのような重要な部分同士が互いにキャッシュから追い出し／追い出されることによるコンフリクトミスを防ぐ必要がある．そのためには，キャッシュ容量程度のサイズのグループ化を行う階層レベルを設けるのがよい．ただし，次節で述べるコピーアルゴリズムによる CC-HC 配置ではグループ内は幅優先順に，つまり，木のルートから特定の深さまでの部分木はまとまって配置されるため，別に階層レベルを設けずともコンフリクトミスを防ぐ形となっている．

4.3 Cache-conscious 階層的グループ化のためのコピーアルゴリズム

Cache-conscious な階層的グループ化の実現のため，**図 9** に示す新しいアルゴリズムを提案する．

手続き `copy_cluster` は，`p0` から移動させられるオブジェクトを先頭オブジェクトとして，`free` ポインタの直後に，階層レベル `level` のグループ（C と呼ぶ）を構成する．

もし `level` が 0 であれば，`copy_cluster` は単純に先頭オブジェクトのみを `p0` から `free` へ通常のコピー GC の要領でコピーする．それ以外の場合，再帰的に構成されるレベル `level`−1 の先頭グループと，それに続けて配置されるレベル `level`−1 の派生グループとで C を構成する．

例えば，**GC ルート**（`root`）は次のようにしてス

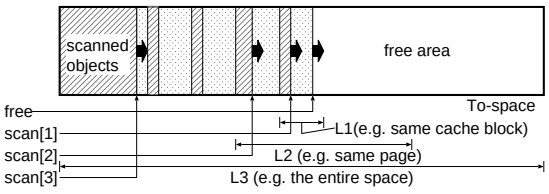


図 8 複数のスキャンポインタを用いた提案する cache-conscious な階層的グループ化アルゴリズムでの To 空間
Fig. 8 To-space of our cache-conscious hierarchical clustering algorithm with multiple scan pointers.

キャンできる：

```
copy_cluster(root, MAX_LEVEL);
root = FWD_PTR(root);
```

ここで，MAX_LEVEL は先に述べた最大の階層レベルであり，配置のポリシーに基づいた定数である．

つまり，図 9 における再帰呼び出し（中央付近の p0 を先頭オブジェクトとした先頭グループを構成するための copy_cluster の呼び出しと，while ループ中の *link を先頭オブジェクトとする派生グループを構成するための copy_cluster の呼び出し）の入れ子の深さは定数であり，再帰呼び出しが消えるまでインライン展開することができる．図 9 では，派生グループの先頭オブジェクトらは while 文の部分で C をグループ内スキャンポインタでスキャンすることによって見つけられる．グループ内スキャンポインタは各 level の手続きの局所変数 scan に保持されているため，インライン展開したもの[⋆1]で考えるとレベル毎のスキャンポインタを持つことになる．

これら複数のスキャンポインタを図 8 の scan[1]，scan[2]，scan[3] のように例示すると，提案するアルゴリズムはスキャンするポインタを切り替えながらコピーを進めるものだとも解釈できる．これは，図 6 に示した Moon のアルゴリズムの一般化に当たり，小さな番号のスキャンポインタが高い優先度を持つ形でスキャンを進めることに相当する．優先度の低いポインタでのスキャンが進められるのは，それより優先度の高いすべてのポインタについて後述する拡張目標を達成した場合（free >= ext）など当面スキャンすべきオブジェクトが存在しない場合に相当する．

関数 extension_target は望ましい配置のポリシーを具体化する．望ましい配置は，各階層レベルがメモリ階層のどの側面の局所性をどのように改善するかで定まる．このため，C を階層レベル level の望ましい**ブロック**あるいは（望ましい境界整列つきでの）望ましい**サイズ**に合わせるために，leader アドレスを起点とするグループ化の望ましい目標（つまり望ましい境界）を計算する．

```
pointer free;      // the free pointer

/* Constuct a cluster of level "level"
   with the leader object at "p0". */
copy_cluster(pointer p0, int level)
{
 pointer leader;  // the leader object
 pointer scan;    // local scan pointer
 scan = leader = free;   // in TO-space
 if(level == 0)
  {
    MOVE(p0, free);
    FWD_PTR(p0) = free;
    free += SIZE(free);
    return;
  }
 // the leader cluster
 copy_cluster(p0, level-1);
 pointer ext;      // extension target
 ext = extension_target(leader, level);
 if (free >= ext) return;
 while (scan < free)
  {
   foreach (link in CHILDREN(scan))
    if (FROM_SPACE(*link))
     if (TO_SPACE(FWD_PTR(*link)))
       *link = FWD_PTR(*link);
     else
      { // offspring clusters
        copy_cluster(*link, level-1);
        *link = FWD_PTR(*link);
        if (free >= ext) return;
      }
   scan += SIZE(scan);
  }
}
```

図 9 cache-conscious な階層的グループ化コピーアルゴリズム
Fig. 9 Cache-conscious hierarchical clustering copying algorithm.

例えば，図 8 の 3 つの区間それぞれをキャッシュブロック，ページ，To 空間全体に対応させるに当たっては，図 8 に示した各区間の右端のアドレスを，各レベルの拡張目標とする．例えば，64B のキャッシュブロックに合わせるには，extension_target(leader, 1) は leader +64 − (leader *mod* 64) を返し，4096B の仮想記憶のページに合わせるには extension_target(leader, 2) は leader +4096 − (leader *mod* 4096) を返すようにする．

これ以外の拡張目標として，仮に L3 グループを 16KB のサイズ（ただし 64B 境界で）に合わせるとした場合は，extension_target(leader, 3) は leader +16384 − (leader *mod* 64) を返すことにする．

⋆1 展開は機械的でプログラムで示すのは冗長なため，論文中では具体的な展開結果を示さない．

5. Cache-oblivious 配置と深さ優先順配置

我々が以前提案した cache-oblivious 階層的グループ化（cache-oblivious hierarchical clustering，CO-HC と略す）配置法[20] では，特定のメモリ階層間転送単位・変換単位に関するサイズやブロック境界を意識することなく，様々なスケールの局所性を持たせた配置とすることで様々なキャッシュ等に自動的・適応的に対応した．CO-HC では，データ構造のグラフ構造そのものを利用し，先に見た図 7 のように，先頭オブジェクトからの距離が 0〜1，2〜3，4〜5，…のオブジェクトを階層レベル L1 としてグループ化し，L1 グループを仮想的なオブジェクトとみたてて同様に L2 としてグループ化する．L3 以上のグループ化も同様である．

CO-HC 配置を実現するコピーアルゴリズムでは，階層レベル毎のスタックを用いた．深さ優先順の配置を実現するためにもスタックは必要であるが，スタックを用いる場合は溢れ対策が必要となる．そのため我々は，スタック溢れの直前にスタックで本来管理する未スキャン部分の情報を Cheney のキューで管理するモードに一時的に移行し，スタック溢れを防ぐ**限定スタック法**を提案した[20]．また，CO-HC 配置用コピーアルゴリズムにおけるスタック溢れ対策に限定スタック法を用いることを提案した[20]．

本論文での性能評価では，深さ優先順の配置にはこの限定スタック法を用い，CO-HC 配置には上で述べた CO-HC 配置用コピーアルゴリズムを用いた．また，深さ優先順と CO-HC の中間として，cache-oblivious な (階層的ではない) グループ化を考えることができる．これは，先頭オブジェクトからの距離が 0〜1，2〜3，4〜5，…のオブジェクトを階層レベル L1 としてグループ化し，L1 グループを仮想的なオブジェクトとみたてて深さ優先順に配置する方式であり，性能評価にも加えた．

6. 実　　装

次節で述べるマイクロベンチマークでコピーアルゴリズムの純粋な効果を測定可能とするために，マイクロベンチマークを記述するための C 言語でのフレームワークを実装した．提案方式を含む各コピーアルゴリズムは C 言語で記述され，明示的な割り当て／GC 用 API と，コピー GC がスキャン可能な GC ルートを保持するポインタスタックが提供される．ミューテータのプログラムをこれらの API を用いて C 言語で作成した．本来はそのようなプログラムは実装の詳細を隠した高水準言語から翻訳・生成されるべきではあるが，ここでは高水準の言語処理系の作成ではなく性能評価を目的として，直接 C プログラムを作成した．

Moon のアルゴリズムと同様に，CC-HC コピーアルゴリズムにも To 空間内に複数のスキャンポインタで重複してスキャンされる部分が存在するという問題がある．その様々な対処は 7.5 節で議論するが，比較的効果がある単純な方法で再スキャンの一部を避けられる．それはある階層レベルで構成したグループがすぐ上の階層レベルの先頭グループであった場合，その階層レベルで最後にスキャンしたところまでのスキャンを，すぐ上の階層レベルでは省略できるという点である．もちろん，先頭グループでない場合は先頭グループと当該グループとの間に未スキャン部分が一般には存在するため省略できない．この再スキャン一部省略法は，図 9 の関数からリターンする箇所にて `scan` の値を（変数などに保存して）返すようにし，中央付近の `p0` を引数とする呼び出しでは，返された直下の階層の `scan` の最後の値で自分の `scan` を更新するようにすればよい．

7. 評　　価

UltraSPARC-IIe をプロセッサとする Sun Blade 100 ならびに Pentium 4 や Xeon をプロセッサとする PC にて性能測定を行った．測定環境・測定条件は**表 1** の通りである．また，キャッシュを意識した CC-HC のためのパラメータは**表 2** の通りである．

表 1 に補足する．評価プログラムのプロセスでは単一プロセッサ（コア）のみを（OS の機能で固定的に）用いた．Xeon ではプリフェッチに関する BIOS オプションが設定できたためハードウェアプリフェッチをオフにした．オンにした場合は性能の低下が見られたが，キャッシュされていた重要なデータが，余計なプリフェッチ結果で置き換えられてしまうためと考えられる．Pentium 4 の PC には BIOS オプションがなかったためオフにできなかった．また，Xeon では隣接キャッシュラインのプリフェッチをオンにした．これにより Pentium 4 におけるセクタ単位（128B）での転送と同じことになる．Xeon では 64 ビット版 OS を用いたが，ほとんどの評価プログラムは Pentium 4 と共通の 32 ビット版とした．ただし，サイズが 4GB を超えるプロセス，あるいは OS に追加したデバイスドライバからプロセッサの性能カウンタを読み出すプロセスのために 64 ビット版プログラムも準備した．

表 2 に補足する．Pentium 4 ではキャッシュのセクタ単位（128B）でグループ化するよりも，64B ブ

表 1 システムパラメータ
Table 1 System parameters.

CPU	UltraSPARC-IIe 500MHz	Pentium 4 3.00GHz HT	Xeon X5460 3.16GHz Quad-Core
L1D cache	16KB 32B-line (two 16-byte sub-blocks) direct-mapped	16KB 64B-line 8way set-associative	32KB 64B-line 8way
L2 cache	256KB 64B-line 4way	1MB 128B-sector (64B-line) 8way	6MB(×2) 64B-line 24way
Prefetcher		Enabled (No BIOS option)	HW prefetch: Disabled, Adjacent line prefetch: Enabled
TLB	64-entry dTLB	64-entry DTLB	16-entry DTLB0, 256-entry 4way DTLB1
VM page	8KB	4KB	4KB
Memomry	256MB	512MB	8GB
OS	Solaris 9	Linux 2.6.25	Linux 2.6.25
CC	GCC 3.2 `-mcpu=ultrasparc -O3 -fno-strict-aliasing`	GCC 4.3.2 `-O3 -fno-strict-aliasing`	GCC 4.3.2 `-O3 -fno-strict-aliasing`

ロックのグループ化と 256B サイズ（128B 境界）のグループ化を組み合わせたほうが性能がよかった．これはハードウェアプリフェッチによりキャッシュの転送単位が擬似的に増加しているためと考えられる．Xeon ではキャッシュや仮想記憶の転送・変換単位となるブロックに加えて階層レベル L4 として 16KB サイズでのグループ化も追加した．これにより TLB ミスは増加することもあるが，L2 キャッシュミスが減少し，総合的にはより高い性能が得られた．L2 キャッシュミスの減少は，ポインタベースのデータ構造において末端オブジェクトが集められる効果で説明できる．つまり，中間部のオブジェクトのグループ化を，末端部が残した中途半端な領域に対して実施せずに済ませられる効果がある．同じ効果は UltraSPARC-IIe でも見られたが，ここでは簡単のため L3 以上は設定しなかった．一方，Pentium 4 で L4 以上を追加した場合は性能が低下した．これには，TLB が Xeon と比較して少ないために L2 キャッシュに関する改善が薄められる点，ならびにページ境界を越えるアクセスも近いアドレスに集められる結果，ハードウェアプリフェッチが働いてしまうことが多くなる点が考えられる．

ポインタベースの大規模データ構造上の探索時間の改善を示すために，3 つのマイクロベンチマークを代表となる評価プログラムとして選択した．すなわち，2 分探索木，2 分探索木配列（要素が 2 分探索木の配列）ならびに連想リスト配列（要素が連想リストの配列）である．ポインタベースのデータ構造を用いた基本的な探索アルゴリズムとしては，リスト上の線形探索，2 分探索木上の探索，チェイン法を用いたハッシュ表上の探索などが知られているが，リスト上の線形探索はチェイン法に含まれること，以下ではランダムなキーについて評価するためハッシュ関数は重要でないことからこのようにした．また，単一のリスト上の線形探索はデータ構造の構築に膨大な時間を要するので除外している．2 分探索木配列は表，木，リストを組み合わせた意味のある探索用データ構造の 1 つとして選択した．

表 2 cache-conscious な階層的グループ化パラメータ
Table 2 Cache-conscious hierarchical clustering parameters.

	UltraSPARC-IIe	Pentium 4	Xeon
L1	64B block	64B block	64B block
L2	8KB block	256B size 128B align	128B block
L3		4KB block	4KB block
L4			16KB size 64B align

以下では主に探索時間の削減に着目して性能測定を行うとともに，その分析のためにいくつかの組み合わせについては L2 キャッシュミス回数と TLB ミス回数を測定した．また GC 時間も測定した．測定では，まず指定された個数のランダムなキーと値のペアを探索用データ構造に登録する．登録完了後に明示的にコピー GC を行う．登録数はコピー GC の後にヒープ中で生きているオブジェクトが占めるサイズが特定の値となるように設定した．その後はランダムなキーで探索を繰り返し，各探索では同じキーを持つ登録済みのペアがあればその値を返すこととする．キーと値には 32 ビット整数を用いた．

7.1 2 分探索木

図 10 に 2 分探索木における探索 1 回あたりの平均探索時間をクロック数に換算して示す．図 10 の横軸において UltraSPARC-IIe と Pentium 4 はそれぞれ SPARC と P4 と略記している．また Xeon については 64 ビット版プログラムを用いた場合について x64 と表記した．またコピー GC による再配置直後の生きているオブジェクトの総量を「_50MB」のように示し

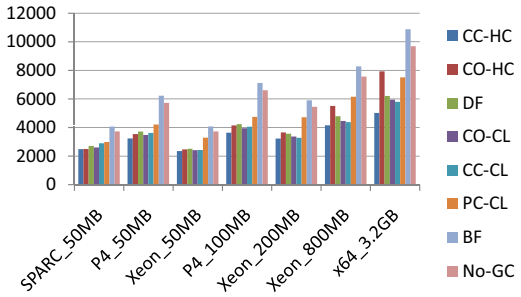


図 10 2 分探索木における探索あたりの平均探索時間（クロック）
Fig. 10 Average search time on binary search tree.

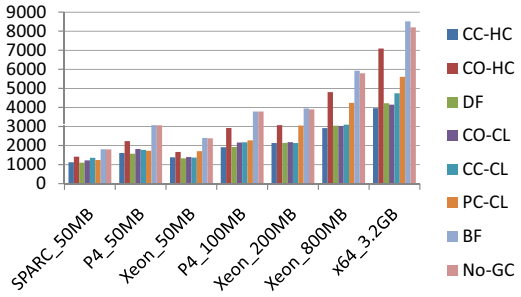


図 11 2 分探索木の配列における探索あたりの平均探索時間（クロック）
Fig. 11 Average search time on array of binrary-search-trees.

た．32 ビット版の場合，2 分探索木のノードオブジェクトのサイズは 20B（64 ビット版では 32B）であるが，8B 境界とするため 24B を使用することになる．

配置を決定するコピー GC アルゴリズムには表 2 による cache-conscious な階層的グループ化を行うもの（CC-HC）と cache-oblivious な階層的グループ化（CO-HC）を用いた．その他にも比較対象として，限定スタック法を用いた深さ優先順（DF），5 章の最後で述べた cache-oblivious なグループ化（CO-CL）も加えている．さらには，CC-HC アルゴリズムをキャッシュに限定して階層的でなく適用したもの（CC-CL）と，同じくページのみを意識した page-conscious なもの（PC-CL）も追加している．CC-CL については SPARC では 64B ブロック，その他では 128B ブロックでグループ化し，PC-CL については仮想記憶のページをブロックとしてグループ化した．図 10 に示した残りの BF と No-GC は，それぞれ幅優先順が得られる Cheney のアルゴリズムとそもそも GC を用いない（再配置しない）ときを表す．

図 10 のすべての場合において提案する CC-HC が最短の探索時間となった．標準的な Cheney のアルゴリズム（BF）と比較して（擬似乱数によるキーの計算時間を含むにもかかわらず）2 倍以上高速化されている場合もある．一方 CO-HC は 50MB のときは CC-HC に次ぐ性能を示しているが，データオブジェクト総量が増えるに従い CC-HC との差が開く結果となった．これは固定サイズのスタックを用いつつ，スタック溢れは Cheney のキューを使って対処するため，その後の探索時にキューを作った層が探索のリンクを辿る過程で「幅優先の段差」として残ってしまうためと考えられる．キューの使用は 100MB で 2 回，200MB で 4 回，800MB で 17 回，3.2GB で 41 回発生していた．対処の 1 つとしてスタックのサイズをヒープに比例させて増やすことが考えられるが，固定サイズの作業空間でどんなヒープでもコピーできるという特徴が失われることになる．なお中間的な案として，CC-HC の要領で再スキャンを用いることでスタック溢れを許容することも考えられる．

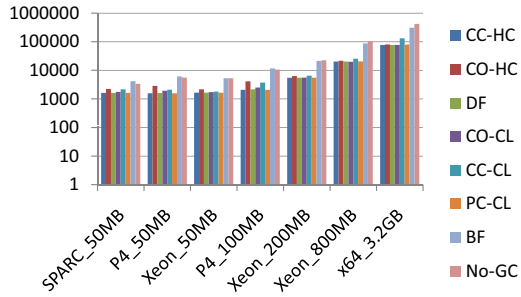
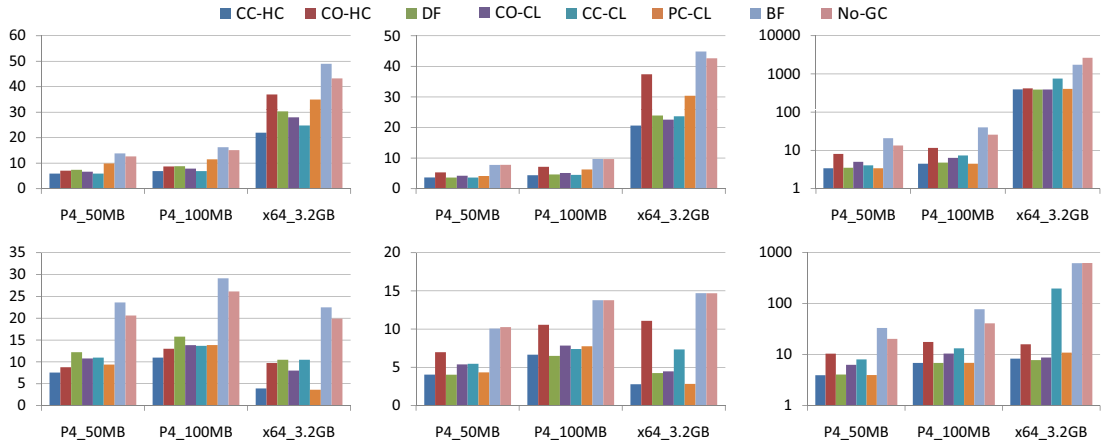


図 12 連想リストの配列における探索あたりの平均探索時間（クロック）
Fig. 12 Average search time on array of alists.

図 10 では主にキャッシュを対象とした CO-CL や CC-CL も健闘しているが，(Xeon では 256 エントリもの DTLB1 があり増えにくくはなっているが）TLB ミスが増えた場合には性能改善は限定的であった．また PC-CL ではキャッシュについての性能改善が得られにくい．L2 キャッシュミス回数と TLB ミス回数については詳しくは後述するが，**図 13** に示している．また深さ優先順（DF）はこれら単一レベルのグループ化と比較しても性能改善は限定的であった．

7.2 2 分探索木配列

図 11 に 2 分探索木配列における探索 1 回あたりの平均探索時間をクロック数に換算して示す．配列は 65536 要素のポインタ（32 ビットまたは 64 ビット）から構成され，キーの特定のビットでエントリが選択されるものとした．2 分探索木のノードオブジェクトは先のマイクロベンチマークと同様である．



(a) 2 分探索木 (binary search tree)　(b) 2 分探索木配列 (array of binary-search-trees)　(c) 連想リスト配列 (array of alists)

図 13 探索における L2 キャッシュミス回数（上段）と TLB ミス回数（下段）
Fig. 13 L2-cache-misses (upper part) and TLB-misses (lower part) for search.

図 11 でもすべての場合において提案する CC-HC が最短にほぼ近い探索時間となった．ただし，特にデータオブジェクト量が少ないときに DF のほうが高速な場合が見られる．前節での 1 つの大きな木構造が 65536 個の小さな木構造に分割されているため，キャッシュに関する局所性改善がより重要であること，DF は木構造の末端付近を集めるのにより効果的であることといった理由が挙げられる．CC-HC はトップダウンにグループ化を決定するため，DF のように，末端まで先にたどり着くことでのボトムアップ的情報を使ってのグループ化はできない．

CO-HC の性能は良くないが，大きな配列が加わったため，CO-HC でのキューの使用は 50MB で 1 回，100MB で 3 回のように増加している．また，グラフの構造のみを見る CO-HC は木のような比較的均質なデータ構造には有効だが，大きな配列のような特異なオブジェクトが加わるとうまくいかなくなる．

7.3 連想リスト配列

図 12 には連想リスト配列における探索 1 回あたりの平均探索時間をクロック数に換算して示した．CC-HC と BF との性能差は場合によっては 5 倍以上に拡大しているため対数グラフとしている．配列は前節の 2 分探索木配列と同じで，連想リストはキーと値のペアとからなる Lisp 的な “cons セル”（16B）とそのリストのための “cons セル”（16B または 64 ビット版では 24B）からなる．

図 12 でもすべての場合において提案する CC-HC が最短にほぼ近い探索時間となった．ただし，似た性能が得られるアルゴリズムも多い．これは，配列部分，リスト部分ともデータ構造に幅がなく，いくつかのアルゴリズムでは似通った配置結果が得られたためと考えられる．ここで CO-HC は配列とリストの先頭の cons セルをルートからの距離のみに基づきグループ化しようとするため，配置に幅優先的な性質が加わってしまったと考えられる．CC-HC ではより具体的なブロックサイズなどの情報を用いるためこのような問題は起こらない．

7.4 ミス回数削減

Linux 2.6.25 に性能測定ツールを組み込んだ環境においては，探索あたりの L2 キャッシュミス回数と TLB ミス回数を測定した．その結果を，図 13 に示す．これから CC-HC は，L2 キャッシュミスと TLB ミスを同時に削減できていることがわかる．つまり，L2 キャッシュミスについては CC-CL に匹敵する回数，TLB ミスについては PC-CL に匹敵する回数へと削減している．

7.5 GC 時間

一方，総データオブジェクトの 1 バイトあたりの GC 時間をクロック数に換算して，**図 14**，**図 15**，**図 16** に示す．これから，(1) CC-HC の GC 時間は比較的長い，(2) DF の GC 時間が比較的短い，(3) BF の GC 時間は連想リスト配列では長くなるがその他では短い，という傾向が見て取れる．DF と BF の GC 時間が比較的短いのは，基本アルゴリズムが単純であるため，プログラムを多少複雑にしてより高速なコードで実装済とした点が挙げられる．つまり，他のアルゴリズムでも複雑だが高速なコードとすることでの GC 時間短縮を目指せる．

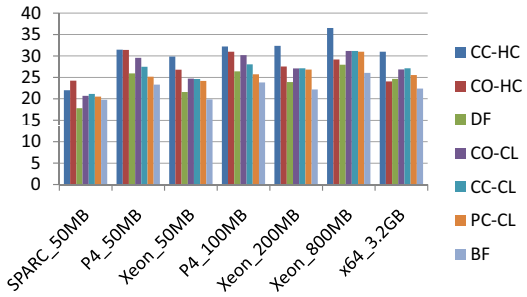


図 14 2 分探索木における GC 時間（クロック／バイト）
Fig. 14 GC time on binary search tree.

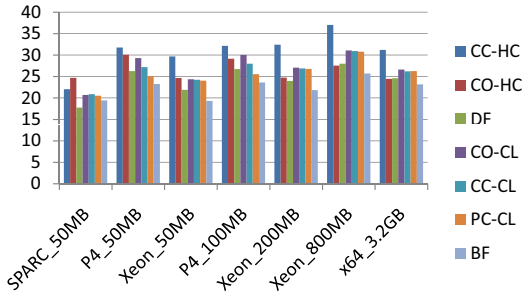


図 15 2 分探索木の配列における GC 時間（クロック／バイト）
Fig. 15 GC time on array of binary-search-trees.

もう 1 つ，CC-HC 特有の問題とて複数回の再スキャンの問題がある．CC-CL や PC-CL は CC-HC アルゴリズムを 1 段のみのグループ化として用いているが，表 2 のように，2 段以上の階層的グループ化とした場合は，その階層レベル用のスキャンポインタが増え，その再スキャンの分だけ GC 時間が延びることになる．この差は図から見てとることができる．この問題へは次の対処が考えられる．

- 直下の階層レベルでスキャン済みの範囲の再スキャンをスキップする．この方法のうち，直下の階層レベルも先頭グループであった場合には単純に最後のスキャンポインタを引き継げばよいことは 6 章で述べたとおりであり実装済みである．一方，より一般の場合については，3 章で述べた hierarchical decomposition[16] を用いることができる．ただし，キャッシュラインのような小さなブロックに適用すると適用対象と比較して多くの作業空間を要する．
- より低いすべての階層レベルでスキャン済みの範囲の再スキャンをスキップする．直下の階層レベルでスキャン済みの範囲よりも，それ以下の階層レベルでスキャン済みの範囲の和集合のほうが一般には大きい．このため上で述べた直下のみを扱うよりも効果は高い．そのためには，未スキャン部分とスキャン済み部分の境界の集合を管理するキューのようなデータ構造を新たに準備し，上位の階層レベルにその情報を引き継ぐとともに，キューが満杯に近くなれば適宜詰め直す方法が考えられる．しかし，そのような方法は提案している CC-HC アルゴリズムの単純さ（開発・デバッグの容易さ）を失わせることになる．あるいは，GC 時間短縮は限定的となるが，To 空間のオブジェクトにスキャン済ビットを持たせて，オブジェクトのスキャン済ビットのスキャンは省略できないが，フィールドスキャンは省略できるとする方式も考えられる．

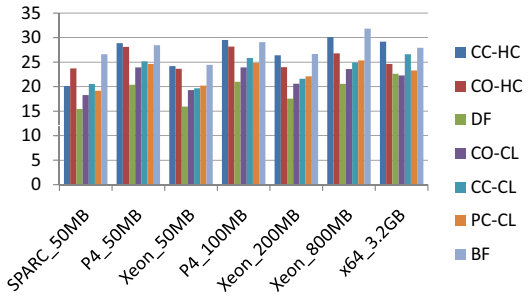


図 16 連想リストの配列における GC 時間（クロック／バイト）
Fig. 16 GC time on array of alists.

8. 関 連 研 究

深さ優先順配置は一般に幅優先順配置よりも局所性を改善するが，スタックのための余分の作業空間を必要とする．5 章で述べた限定スタック法[20] は，限られた作業空間で深さ優先順をできるだけ近似するものであったが，他にも，From 空間の未使用部分をうまく使うことで余分の作業空間を使わない方式[14),19)] も提案されている．

「グループ化」については，近年，Siegwart と Hirzel が，仮想記憶レベルの局所性改善のための「グループ化」アルゴリズム[16] を並列コレクタ向けに拡張し，製品レベルの JVM での評価を行っている[13]．彼らはキャッシュレベルを含む階層的グループ化は行っていないが，26 ベンチマークのうち 14 個のベンチマーク，特に SPECjbb2005 と SPECjvm98 `db` において速度向上を得ている．

我々のアルゴリズムはデータオブジェクトの「静的グラフ構造」を用いて階層的グループ化を行う．同様に「静的グラフ構造」を用いるものは多数，研究され

ているが[5),10),16)]，他の静的情報，あるいは実行時情報が得られればより適切な配置を行うことができる．静的情報としては型情報を利用するものが提案されている[9),12)]．文献 12) は「多産な型」の概念に基づくとともにオフラインのプロファイル情報も用いて割り当て時と GC 時に局所性のよい配置を実現する．一方，実行時情報を用いて GC 時に再配置を行うもの[3),6),8)]では低オーバヘッドでの実行時プロファイリングが課題となる．このうち文献 3) では，GC とは別に積極的な局所性改善を行うことを提案しており，アクセスバッファに基づく親和性解析による深さ優先順キャッシュレベル局所性改善と，参照ビットに基づき選択したオブジェクトらの hierarchical decomposition[16)] によるページレベル局所性改善を同じシステムで組み合わせている．ただし，これらは異なるオブジェクトに適用されるため，本論文で提案するような階層的グループ化によるキャッシュと仮想記憶の同時改善とは異なる．その他の追加情報としては，ユーザからの付加情報を用いること[11)] も提案されている．

実行時のアクセスを直接のトリガとして局所性を改善する手法も提案されている．White は大規模 Lisp システムにおいてアクセスされたオブジェクトを（Lisp マシンのハードウェア/マイクロコード機構を用いて）近くのページに移動させること[15)] を 1980 年の時点で提案している．同様の方式として，ミューテータのアクセス時に Baker の実時間コピー GC のリードバリアの要領で，アクセスされたオブジェクトをそれらを集める空間にコピーする方式[7)] が提案されている．

Cache-oblivious 配置については，B-tree のような特定のデータ構造に対する理論的研究[1)] が知られている．そこで提案されている配置は基本的に図 7 に等しいが，我々の CC-HC コピー GC 方式では特定のデータ構造だけでなく GC が対象とするあらゆるデータ構造に対してそのサイズやポインタ数などに応じた cache-conscious な階層的グループ化がなされる．また，Zhang らは cache-oblivious な階層的グループ化において，参照親和性モデルに基づいて各階層レベルのグループをボトムアップに決定する方法[18)] を提案している．

Cache-conscious なデータ配置では本研究が主に対象としている容量ミス・義務的ミスの削減以外に，連想度の問題からのコンフリクトミスの削減を対象とすることがある．プロファイリングに基づきコンパイラがコンフリクトミスを削減するような配置（オフセット）を求める方式[2)] が提案されている．

9. おわりに

本論文では，我々が提案している「階層的グループ化」において，cache-conscious な配置を実現するコピーアルゴリズムを提案した．階層的グループ化はデータオブジェクトを複数の階層レベルでグループ化することでメモリ階層でのキャッシュと仮想記憶の双方における局所性を改善する．

提案するコピー GC アルゴリズムでは，複数のスキャンポインタを用い，それぞれがメモリ階層の 1 側面の局所性を高めることで，限られた作業空間のみで cache-conscious な配置を実現できた．本アルゴリズムは特にポインタベースの大規模データ構造上の探索時間を短縮するのに有効であることを，3 つの代表的なマイクロベンチマークを用いて示した．

今後は，本コピーアルゴリズムを，Java や Lisp の処理系に組み込んで評価したい．特に，世代別ごみ集めにおいて中期的な寿命を持つオブジェクトに対しても局所性改善効果が得られるか注目していきたい．

謝辞　本研究の一部は，情報爆発に対応する高度にスケーラブルなソフトウェア構成基盤（18049015）（科学研究費特定領域研究「情報爆発時代に向けた新しい IT 基盤技術の研究」）の補助を得て行った．

参考文献

1) Bender, M. A., Demaine, E. D. and Farach-Colton, M.: Cache-Oblivious B-Trees, *SIAM Journal on Computing*, Vol.35, No.2, pp.341–358 (2005).
2) Calder, B., Krintz, C., John, S. and Austin, T.: Cache-Conscious Data Placement, *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pp.139–149 (1998).
3) Chen, W.-k., Bhansali, S., Chilimbi, T., Gao, X. and Chuang, W.: Profile-guided Proactive Garbage Collection for Locality Optimization, *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pp.332–340 (2006).
4) Cheney, C.J.: A Nonrecursive List Compacting Algorithm, *Communications of the ACM*, Vol.13, No.11, pp.677–678 (1970).
5) Chilimbi, T.M., Hill, M.D. and Larus, J.R.: Cache-Conscious Structure Layout, *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pp.1–12 (1999).

6) Chilimbi, T.M. and Larus, J.R.: Using Generational Garbage Collection to Implement Cache-Conscious Data Placement, *Proceedings of the International Symposium on Memory Management*, pp.37–48 (1998).
7) Courts, R.: Improving Locality of Reference in a Garbage-Collecting Memory Management System, *Communications of the ACM*, Vol.31, No.9, pp.1128–1138 (1988).
8) Huang, X., Blackburn, S.M., McKinley, K.S., Moss, J. E. B., Wang, Z. and Cheng, P.: The Garbage Collection Advantage: Improving Program Locality, *In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp.69–80 (2004).
9) Lam, M.S., Wilson, P.R. and Moher, T.G.: Object Type Directed Garbage Collection To Improve Locality, *Proc. of the International Workshop on Memory Management*, Lecture Notes in Computer Science, Vol.637, pp.404–425 (1992).
10) Moon, D.A.: Garbage Collection in a Large Lisp System, *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pp.235–246 (1984).
11) Novark, G., Strohman, T. and Berger, E.D.: Custom Object Layout for Garbage-Collected Languages, Technical Report UM-CS-2006-06, University of Massachusetts, UMass Amherst (2006).
12) Shuf, Y., Gupta, M., Franke, H. and Singh, J.P.: Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times, In the 2002 ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '02, ACM Press, pp.13–25 (2002).
13) Siegwart, D. and Hirzel, M.: Improving Locality with Parallel Hierarchical Copying GC, *Proceedings of the 2006 International Symposium on Memory Management (ISMM '06)*, pp.52–63 (2006).
14) Thomas, S.P. and Jones, R.E.: Garbage Collection for Shared Environment Closure Reducers, Technical Report 31–94, University of Kent and University of Nottingham (1994).
15) White, J.L.: Address/Memory Management For A Gigantic LISP Environment or, GC Considered Harmful, *LFP'80: Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, pp.119–127 (1980).
16) 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 (Proceedings of PLDI'91), pp.177–191 (1991).
17) Yasugi, M. and Yuasa, T.: Improving Search Speed on Pointer-Based Large Data Structures Using a Hierarchical Clustering Copying Algorithm, *Post-proceedings of the International Workshop on Innovative Architecture for Future Generation Processors and Systems 2007 (IWIA 2007)*, pp.43–52 (2007).
18) Zhang, C., Zhong, Y., Ding, C. and Ogihara, M.: A Method for Hierarchical Data Placement, Technical report, UR Research [http://dspace.lib.rochester.edu/oai/request] (United States) (2004).
19) 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).
20) 八杉昌宏，伊藤智一，小宮常康，湯淺太一：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，情報処理学会論文誌：プログラミング，Vol.45, No.SIG5 (PRO21), pp.36–52 (2004).