平成 24 年度 修士論文

遅延スイープ法における
メモリアクセスの局所性改善

電気通信大学 大学院情報システム学研究科
情報システム基盤学専攻
1153008　金野 千顕

指導教員　小宮 常康　准教授
多田 好克　教授
近藤 正章　准教授

提出日　平成 25 年 2 月 22 日

目次

図目次

第 1 章

序論

ガベージコレクション (GC : Garbage Collection) とは，プログラミング言語における自動メモリ管理機構であり，プログラムが二度と使用しないデータ (ごみ) を見つけて再度利用するための機能を持つ．

一般的なガベージコレクションとして，マークスイープ法 (Mark Sweep GC) がある．このガベージコレクションはルートから生きているオブジェクトを確認するマークフェーズと，ヒープ領域内からごみを回収して再利用するスイープフェーズがある．その回収には，フリーリストと呼ばれるリストに保存する．アロケーションは，フリーリストの先頭のごみを新たなオブジェクトとしてミューテータに渡してヒープ領域に配置する．しかし，親子関係の (連続して参照される) オブジェクトがヒープ領域上の離れた位置に配置されてしまうことも多く，アクセスに時間がかかる．また，ヒープサイズが大きくなるほど，ミューテータ (アプリケーション) の停止時間が長くなってしまう．

一方，マークスイープ法の一種である遅延スイープ法 (Lazy Sweep GC)[1] はマークスイープ法と同様な動作を行なう．しかし，スイープフェーズが少しずつ必要に応じて進められる．ミューテータにより要求されたサイズのごみをメモリ要求時に探すため，ごみの回収は分割して行なわれる．1 つのポインタ (スイープポインタ) を利用して，ヒープをアドレス順に徐々に参照する．これにより，マークスイープ法と比べてガベージコレクションの停止回数が多くなるが，1 回の停止あたりの停止時間を短くすることができる．それにより時間的局所性が多少向上できた．しかし，遅延スイープ法はマークスイープ法の空間的局所性を改善できていないため，さらにメモリアクセスの局所性を向上させることが望まれる．

キャッシュはライン単位で管理されるため，新たに割り当てるオブジェクトは最近参照したヒープ領域の近辺に配置できると良い．しかし，他のオブジェクトが既に割り当てられていて探すのが困難な可能性がある．本研究では親子関係のあるオブジェクトを生成し始めるときに，ラインサイズのメモリ領域 (以後，チャンクと呼ぶ) を割り当て，そこへ関係のあるオブジェクトを配置できるようにする手法を示す．その手法では，後に親子関係のあるオブジェクト群を生成する際は，専用のコンストラクタを明示的に用いるようにしている．このコンストラクタは親子関係のオブジェクトを参照の局所性に考慮してチャン

クに配置する．

本論文の構成は以下のとおりである．第 2 章で背景を述べ，キャッシュや GC について紹介する．第 3 章で関連研究について述べる．第 4 章で提案手法を説明し，第 5 章で本手法の設計について述べる．第 6 章で本手法の実装について，そして，第 7 章で実験と考察について述べる．最後に，第 8 章でまとめと今後の予定について述べる．

第 2 章

背景

ガベージコレクションとミューテータの処理では，以下で述べるように，メモリアクセスの局所性が低下してしまう．

2.1 メモリアクセスの局所性

2.1.1 参照の局所性

一般にプログラムは以下の 2 つの局所性に基づいてヒープ内を参照する．

- **時間的局所性**：あるオブジェクトが参照されたとき，再びそのオブジェクトが参照される可能性が高い性質
- **空間的局所性**：あるオブジェクトが参照されたとき，その近辺のオブジェクトが参照される可能性が高い性質

上記の局所性を利用して，図 2.1 のように時間経過でプログラムの参照場所が徐々に変わっていく．色の濃い場所はプログラムが最近参照した場所である．時間が経つにつれて，色が薄くなることでその場所が参照されなくなってくることを示している．参照された場所はキャッシュラインに格納される．時間的局所性がある場合は再び色のある場所を参照する可能性が高い．また，空間的局所性がある場合は色のある場所の近辺を参照する可能性が高い．

2.1.2 キャッシュの仕組み

最近アクセスされたデータは，図 2.2 のようにキャッシュメモリに置かれる．キャッシュメモリは CPU とメモリの間にあり，この間の転送速度を速くするために使用される．プログラムは CPU の演算処理を利用してデータや情報を処理する．CPU によって計算されたデータや情報を一時的に保存する場所がキャッシュメモリである．キャッシュメモリにデータや情報を置くことで，CPU が再びアクセスされるデータや情報をキャッシュメモリから取り出してくる．それにより，アクセスを高速に行なうことができる．こ

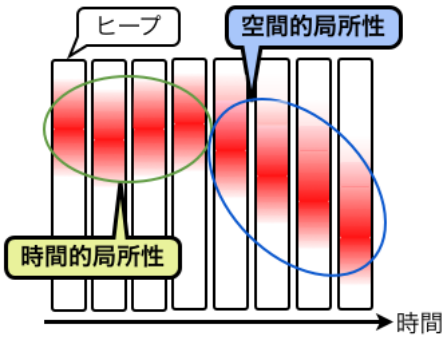

図 2.1　プログラム実行における参照の局所性

れをキャッシュヒットと呼ぶ．

しかし，キャッシュメモリはメモリよりも容量が遥かに少ない．そのため，格納されるデータが限られている．それにより，キャッシュメモリから取り出すときに，最近アクセスされたデータや情報がない場合，メモリから探すことになる．これをキャッシュミスと呼ぶ．

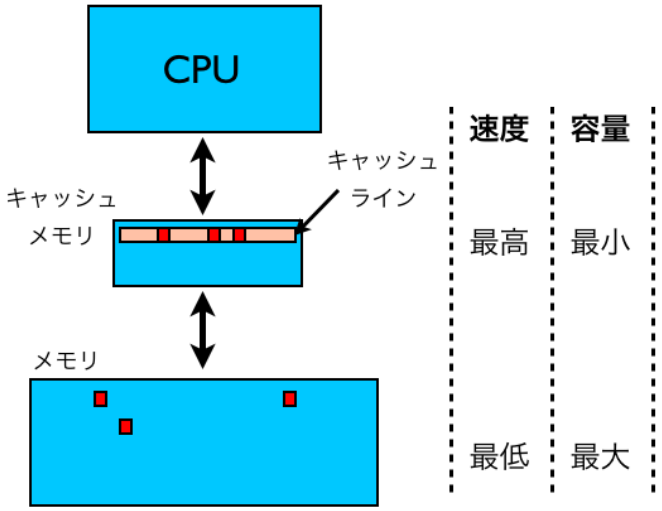

図 2.2　記憶階層の基本構造

キャッシュメモリから取り出す場合，ライン単位でデータを取り出す．そして，キャッシュメモリのデータ格納構造は以下の方式を採用している．

- **ダイレクトマップ方式 (図 2.3)**：メモリにあるアドレスに基づいて，それをキャッシュに格納する位置を割り当てる方式
- **フルアソシアティブ方式 (図 2.5)**：メモリブロックをキャッシュメモリ内の任意の

位置に割り当てる方式

- **セットアソシアティブ方式 (図 2.4)**：上記 2 つを中間にしたもので，連続したキャッシュブロックをセットとしてまとめ，その中であればどこのブロックでも格納できるようにした方式

図 2.3–2.5(文献 [13] から引用) のタグはキャッシュラインにデータがあるかどうかを判別するためのアドレス情報である．タグに探していたデータがすでにキャッシュにあるデータなら，キャッシュヒットをする．もしない場合は，キャッシュミスになる．

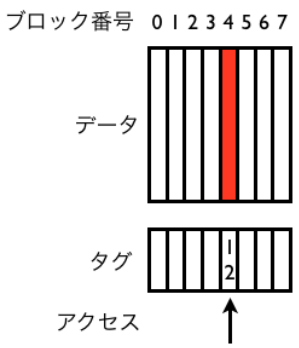

図 2.3 ダイレクトマップ方式

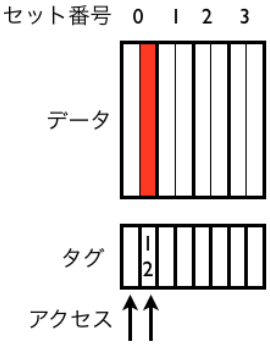

図 2.4 セットアソシアティブ方式

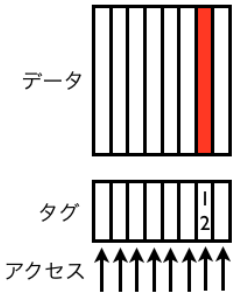

図 2.5 フルアソシアティブ方式

キャッシュには，CPU に近い側から L1(レベル 1) キャッシュ，L2(レベル 2) キャッシュがある．容量は L1 より L2 のほうが大きいが，アクセス速度は L1 のほうが速い．ラインを介してデータを取り出す際には，Intel Pentium4 以降のプロセッサでは L1 キャッシュのラインサイズ 64Byte を取り出す．アクセス速度を向上するにあたって，ラインサイズでキャッシュを取り出すため，空間的局所性が重要になってくる．

2.2 ガベージコレクション

ガベージコレクションについて，参考書 [11] を引用して説明する．

2.2.1 概要

ガベージコレクションとは，プログラミング言語における自動メモリ管理機構である．プログラムが二度と使用されないデータ (ごみ) を見つけて再度利用し，メモリ領域で不要になった領域を解放するための機能である．ごみの定義は実行中のプログラムのどのポインタでも辿りつけないオブジェクトを指す．到達可能である (生きている) オブジェクトはマークをつけて，回収しないようにする．

2.2.2 ヘッダとフィールド

ヘッダは，オブジェクト自体の情報を保持する部分である．主に，オブジェクトのサイズや種類を含まれるが，マークスイープ法ではマークビット (生きているオブジェクトかごみかどうかを判別する機能) の管理に使われる．

フィールドとはオブジェクトがアクセスする可能なデータのことである．フィールドはポインタを持ち，データや他のオブジェクトにアクセスするときに利用する．ポインタは GC の中で重要な役割を持ち，ポインタが正しいメモリ領域を指しているか，正しいオブジェクトを指しているかなど，生きているかごみかを判別するために重要になってくる．また，言語処理系では図 2.6 のとおりに，ポインタを利用してあるオブジェクトから子オブジェクトを指している．これも生きているオブジェクトを探すために重要になってくる．

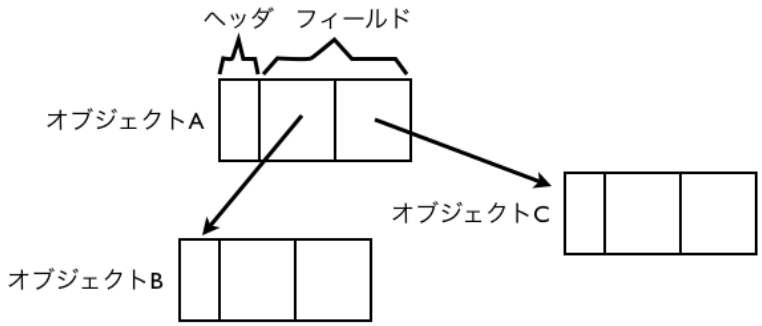

図 2.6 オブジェクトの構造と役割

2.2.3 ミューテータ

ミューテータとは，Dijkstra によって考案された言葉である．その実体は「アプリケーション」と呼ぶ．主に，オブジェクトの生成やポインタの更新を行なう．

2.2.4 ヒープ領域

ヒープ領域とはプログラム実行時にオブジェクトを配置するためのメモリ領域である．ミューテータがオブジェクトのアロケーション (割り当て) 要求をすると，このヒープ領域から必要なサイズのメモリ領域を渡す．GC は，このヒープ領域から生きているオブジェクトをマークして，ごみのオブジェクトを回収する．

2.2.5 ルート

ヒープ領域から生きているオブジェクトを探すには，ルートが必要である．ルート (図 2.7) は大域変数や局所変数である．GC の世界では，オブジェクトを探すための起点になる．GC が開始されると，このルートを辿っていくことで，生きているオブジェクトを探す．生きているオブジェクトにはマークがつけられる．マークのついていないオブジェクトは，ごみとみなされ，GC によって回収される．

プログラミング言語処理系においてルートを参照するときには，スタックを介して行なう．スタックにはローカル変数の配列や処理対象のデータを保持するため，これらがもし生きているオブジェクトを持つなら，辿る必要がある．

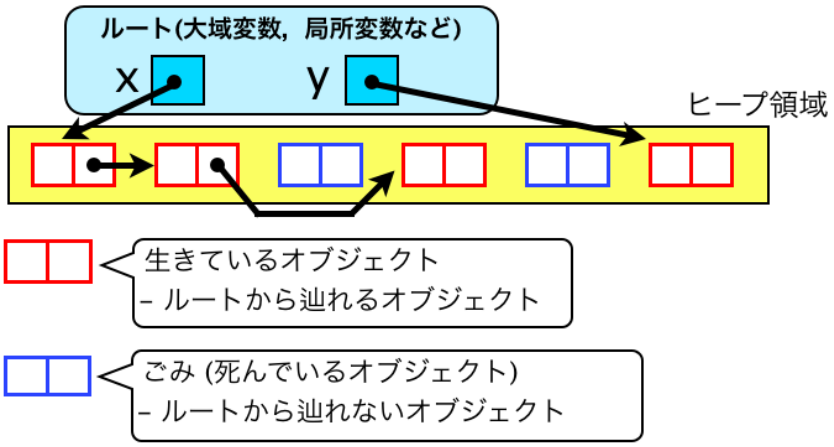

図 2.7 ルートの役割

2.2.6 アロケーション

アロケーション (allocation) とは，ミューテータがヒープ領域に新たなオブジェクトを割り当てる (配置する) ときに，必要なサイズのオブジェクトを割り当てることである．GC の種類によってはアロケーション方法が異なる．今回の手法では重要な役割を持つ．

2.2.7 チャンク

チャンクとは，固まりという意味である．GC の世界では複数のオブジェクトを持つ固まりを示す．ヒープ領域も大きなチャンクである．今回の手法ではキャッシュラインサイズのチャンクを利用する．

2.3 マークスイープ法

2.3.1 概要

マークスイープ法は，ルートから生きているオブジェクトを確認するマークフェーズ (図 2.8–2.9) と，ヒープ内からごみを回収して再利用するスイープフェーズ (図 2.10–2.12) がある．

マークフェーズでは，ルートからオブジェクトを辿っていき，辿れるオブジェクト (生きているオブジェクト) に印をつける．

スイープフェーズは，ポインタがオブジェクトを 1 つずつ参照して，ごみの回収を行なう．また，生きているオブジェクトを参照した場合には印を消す．印のないオブジェクト (未使用オブジェクトやごみのオブジェクト) には，フリーリストと呼ばれるリストに保存する．

この GC 時間にはマークフェーズとスイープフェーズの時間がかかる．マークフェーズでは，生きているオブジェクトの総数に比例した時間がかかる．スイープフェーズではヒープ全体をスキャンするため，ヒープサイズに比例した時間がかかる．

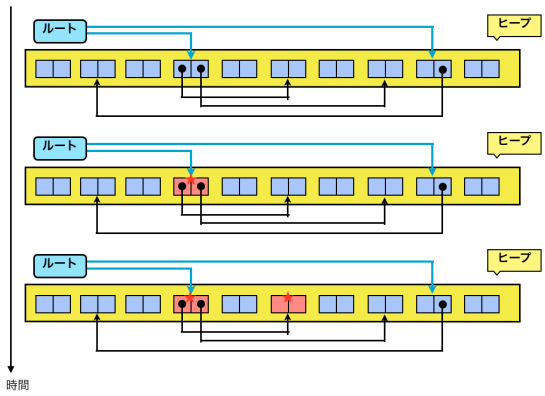

図 2.8 マークスイープ法のマークフェーズにおける処理過程 1

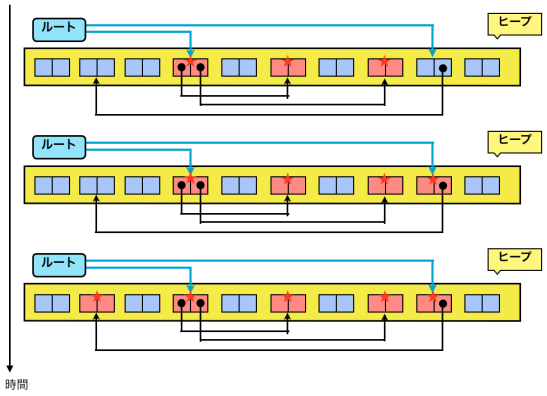

図 2.9 マークスイープ法のマークフェーズにおける処理過程 2

2.3.2 アロケーション

先ほどのスイープフェーズから回収したごみは，新しいオブジェクトを作るために利用される．アロケーションを行なうときには，フリーリストからオブジェクトを取って，新しいオブジェクトをヒープに割り当てる (図 2.13–2.15)．フリーリストはごみがあったヒープのアドレスを記憶しているため，新しいオブジェクトの配置はごみがあった場所に

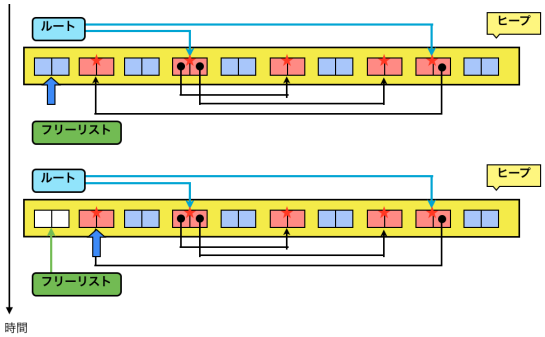

図 2.10 マークスイープ法のスイープフェーズにおける処理過程 1

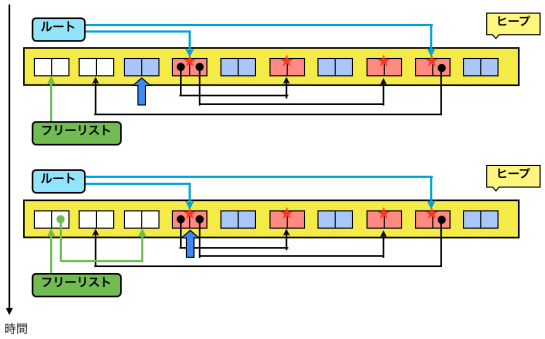

図 2.11 マークスイープ法のスイープフェーズにおける処理過程 2

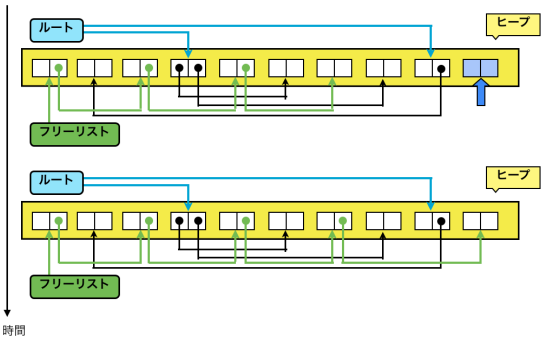

図 2.12 マークスイープ法のスイープフェーズにおける処理過程 3

割り当てられる．フリーリストのオブジェクトがなくなると，再びマークフェーズが始まる (図 2.15)．

2.3.3 問題点

マークスイープ法がプログラムで数回実行されるとチャンクが細分化されていく．そして，ヒープ上には無数のチャンクが点在してしまい，フラグメンテーションが起こりやすい．それにより，親子関係のあるオブジェクトがヒープ領域上の離れた位置に配置されてしまい，処理速度が遅くなり，アクセスに時間がかかる．キャッシュライン上のオブジェクトを参照するため，空間的局所性が利用できない．

また，図 2.16 で示すように，ヒープサイズが大きい場合，スイープフェーズの時間が長くなる．そのためミューテータやアロケーションが動作できない．

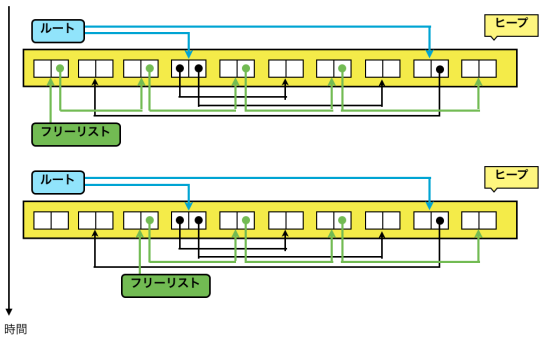

図 2.13　マークスイープ法のアロケーションにおける処理過程 1

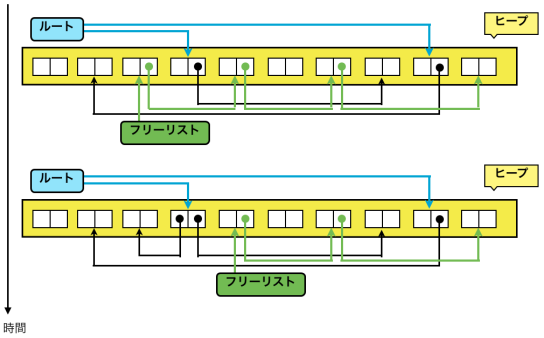

図 2.14　マークスイープ法のアロケーションにおける処理過程 2

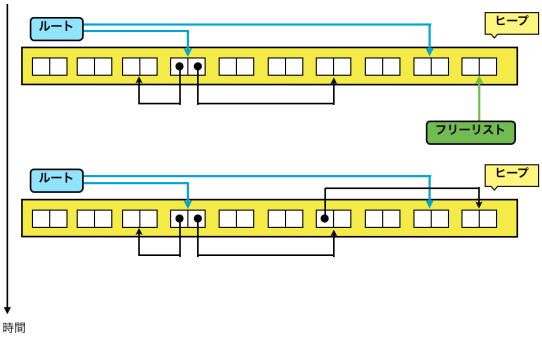

図 2.15　マークスイープ法のアロケーションにおける処理過程 3

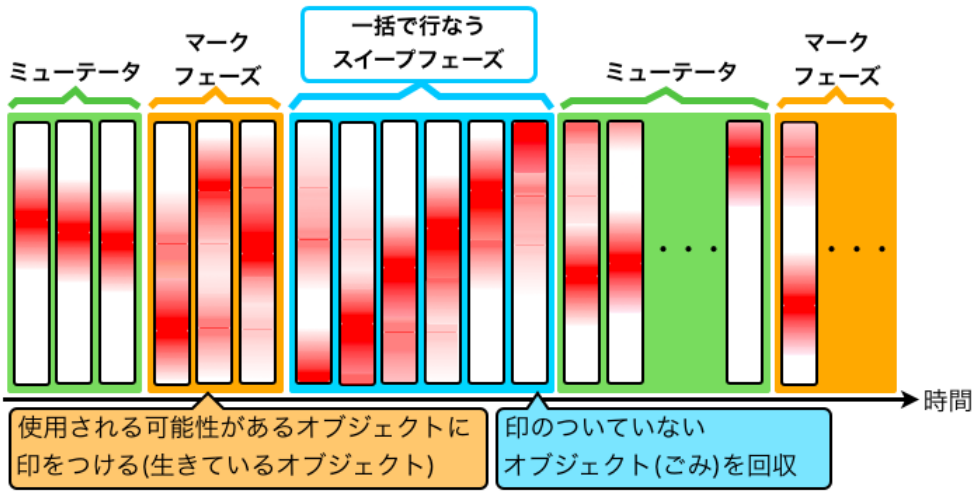

図 2.16　マークスイープ法における局所性

2.4 遅延スイープ法

2.4.1 概要

ガベージコレクションの一種である遅延スイープ法 (Lazy Sweeping)[1] はマークスイープ法と同様な動作を行なうが，スイープフェーズが少しずつ必要に応じて進められる．マークフェーズはマークスイープ法と同じである．スイープフェーズはアロケーションの処理もあり，ミューテータにより要求されたサイズのごみをメモリ要求時に探すため，ごみの回収は分割して行なわれる．つまり，ヒープ領域をアドレス順に徐々に参照する．そのため，図 2.17 のようにミューテータの一時停止を短くすることができる．

そして，色の薄い場所を参照することができるため，多少は時間的局所性が改善する．

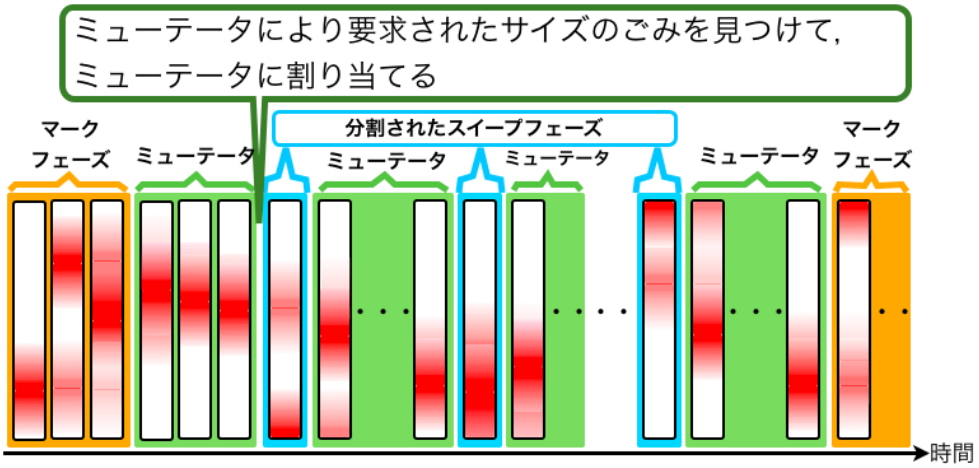

図 2.17　遅延スイープ法における局所性

2.4.2 アロケーション

スイープフェーズ (薄い青の範囲) はアロケーションの処理もあり，ミューテータ (薄い緑の範囲) により要求されたサイズのごみを探す (図 2.18–2.19)．ごみを探すには，スイープポインタと呼ばれるポインタを使って回収を行なう．また，生きているオブジェクトを参照した場合には印を消す．スイープポインタがヒープの終わりまで到達すると，マークフェーズを開始する．オブジェクトの配置はマークスイープ法のフリーリストと同じである．

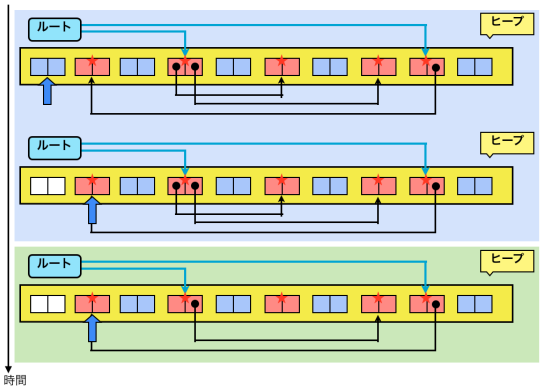

図 2.18　遅延スイープ法のアロケーションにおける処理過程 1

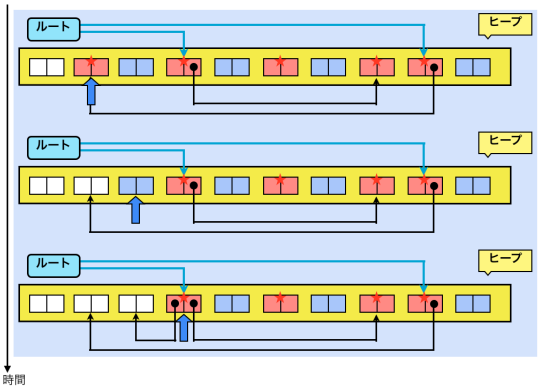

図 2.19　遅延スイープ法のアロケーションにおける処理過程 2

2.4.3　問題点

マークスイープ法と同様に遅延スイープ法のアロケーションにより，親子関係のあるオブジェクトがヒープ領域上の離れた位置に配置されてしまい，処理速度が遅くなり，アクセスに時間がかかる．また，空間的局所性が改善していない．本研究では，このスイープ法の局所性を向上させる手法を示す．

第 3 章

関連研究

3.1 ソースコードからキャッシュミスを減らす手法

文献 [4] の研究ではソースコードからホットループを特定し，オブジェクトの配置の最適化を行なっている．本研究においても，オブジェクトの配置の最適化が役に立つ．

この手法は，ハードウェア性能のモニタ (HPM) に頼らず，キャッシュミスの発生源を特定する．ホットループでのポインタのデリファレンスが Java プログラムでキャッシュミスの主要な源であることを示している．彼らの実験則 (ヒューリスティクス) に基づいた手法では，特定のイディオムのコードパターンに一致するホットループを見つけることで，キャッシュミスを頻繁に引き起こすオブジェクトを識別する．SPECjbb2005 と SPECjvm2008 を使用して，HPM からキャッシュミスの分析結果を調査することで，これらのパターンを特定した．

基本パターンコード (ソースコード 3.1) は，多くのキャッシュミスを引き起こすパターンの一例である．このパターンのホットループでは `objA.field1` をアクセスする際にキャッシュミスを起こす傾向がある (図 3.2)．

ソースコード 3.1 基本パターンコード

```
1  ClassA objA;
2  ClassB objB;
3  while (!end) {  // in a hot loop
4       ...
5       // 1) first, load a reference of ClassA
6       objA = objB.referenceToClassA;
7       ...
8       // 2) then, access a field of objA
9       access to objA.field1;
10      ...
11 }
```

このように親子関係のあるオブジェクトを同じラインに配置するために，オブジェクトの配置の最適化を図 3.3–3.6 に示す．2 つ以上のフィールドを保持するオブジェクトは同じライン上に配置するようにする (図 3.3–3.4)．それらのフィールドが同じラインにあるため，オブジェクトにあるすべてのフィールドに効率よくアクセスすることができる．

また，図 3.5–3.6 では，objB と objA が親子関係の場合，2 つのオブジェクトを同じラ

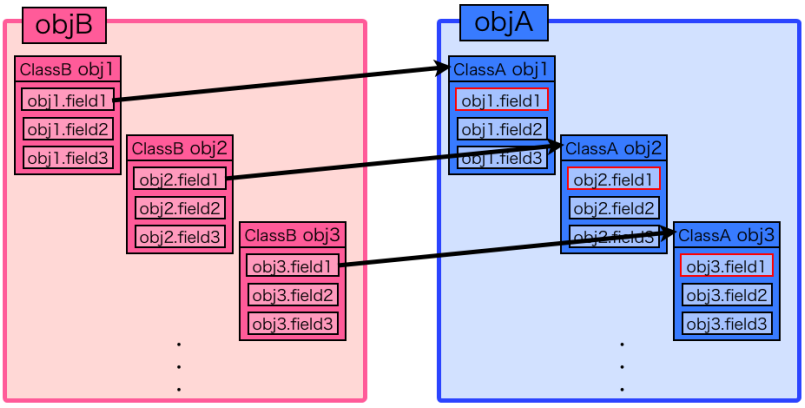

図 3.1　基本パターンコードによるオブジェクトの参照

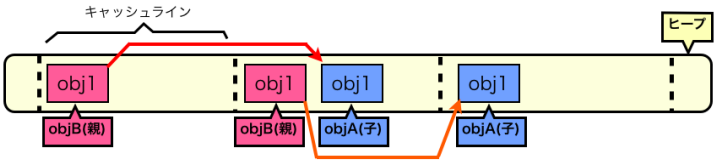

図 3.2　基本パターンコードによるキャッシュライン上のオブジェクト配置

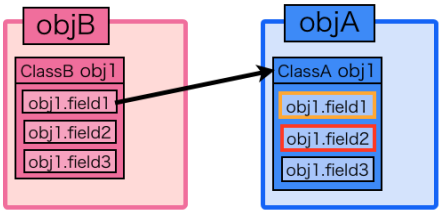

図 3.3　複数のフィールドの参照

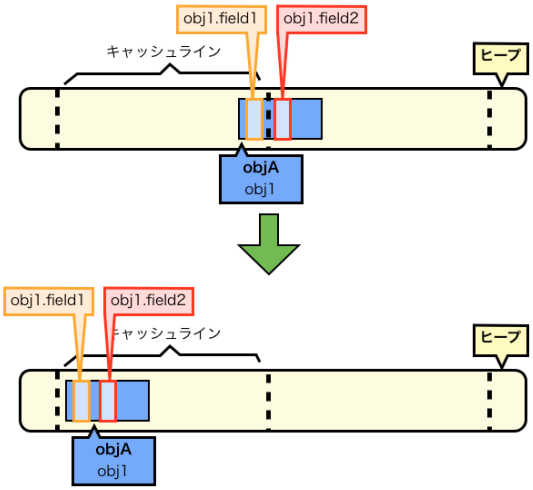

図 3.4　オブジェクトのフィールドを同じキャッシュライン上に配置する処理過程

イン上に配置する．親子関係のオブジェクトが同じラインにあるため連続してアクセスすることができる．

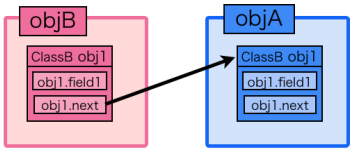

図 3.5　親子関係のオブジェクトの参照

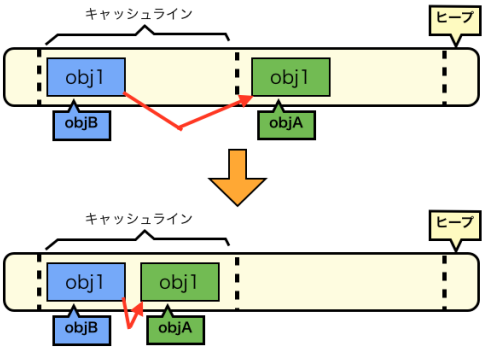

図 3.6　親子関係のオブジェクトをキャッシュライン上に配置する処理過程

3.2　プリフェッチ命令を利用した GC

文献 [6] の研究では，プリフェッチ命令を利用したマークスイープ法を紹介している．

彼らの手法はプリフェッチオングレーと呼ばれ，プリフェッチ命令を利用して GC のマークフェーズを記述することができる．各オブジェクトは関連付けられたカラーがある．白色のオブジェクトはまだ検索していないことを示している．灰色のオブジェクトは検索していることを示す．しかしそのオブジェクトの内部ポインタがまだ検査されていない．そして，黒色のオブジェクトはすで検索し終えて，追跡し終えたポインタが含まれることを示す．灰色のオブジェクトがマークスタックにプッシュされると，すぐにそのオブジェクトの最初のキャッシュラインをプリフェッチする．つまり，灰色のオブジェクトにある内部ポインタは「プリフェッチ命令」を利用することで，次に参照されるオブジェクトを予測することができる．

3.3　コピー法

コピー法 (Copy GC) には，幅優先順と深さ優先順のどちらかに沿ってコピーを行なう．

まず，文献 [7] によるコピー法はヒープ領域を 2 つの semispace に分割する．コピー前 (図 3.8) は片方の semispace にオブジェクトが配置されていて，もう配置できないときにこのコピー法が動作する．そして，深さ優先順 (図 3.7) にコピーを行なう．コピー後 (図 3.9) はもう片方の semispace に生きているオブジェクトが移動する．それにより，アルゴリズムが再帰的になる．また，キャッシュメモリとの相性が良く，データが近いので空

間的局所性が高まる．しかし，ヒープ領域の片方だけでオブジェクトを配置するためヒープ領域の使用率が悪い．そして，再帰的なアルゴリズムによりコピーが行なわれるたびに関数呼び出しが発生し，オーバーヘッドが無視できなくなる．

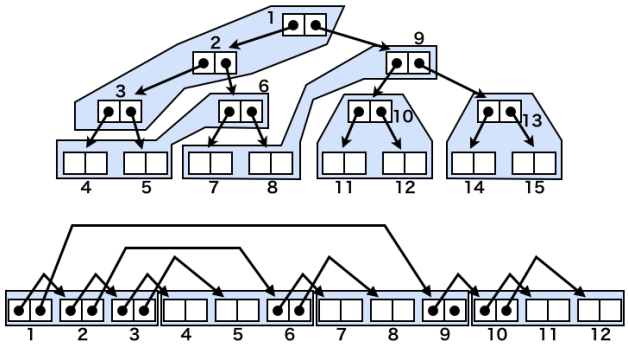

図 3.7　深さ優先アクセス

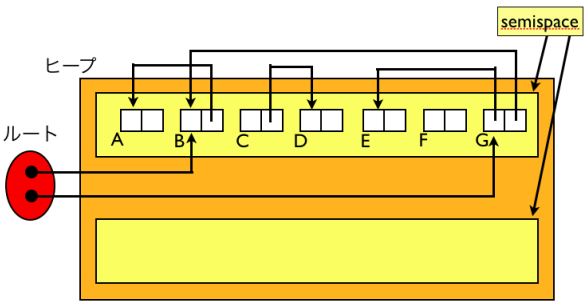

図 3.8　深さ優先順によるコピー法 (コピー前)

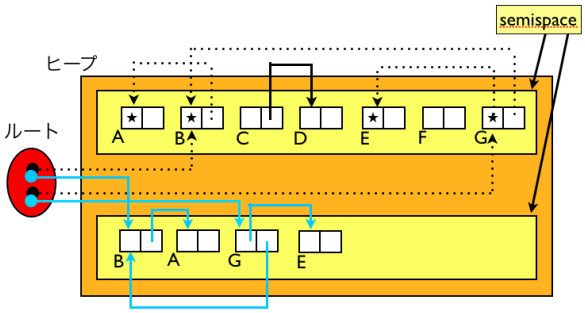

図 3.9　深さ優先順によるコピー法 (コピー後)

文献 [8] によるコピー法においても深さ優先順によるコピー法と同様に 2 つの semispace を用いる．コピー前 (図 3.11) は片方の semispace にオブジェクトが割り当てられないときに，コピー法が動作する．そして，幅優先順 (図 3.10) にコピーを行なう．コピー後 (図 3.12) はもう片方の semispace に生きているオブジェクトが移動する．アルゴリズムは非再帰的であるため，関数呼び出しのオーバーヘッドやスタックの消費を抑えることができる．ヒープ領域がキューのように働くため，探索するための余計なメモリが必要ない．しかし参照関係のオブジェクトが隣り合わないため，キャッシュメモリとの相性が悪くなる．

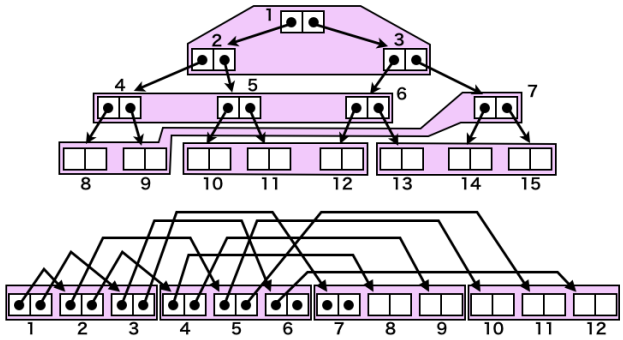

図 3.10　幅優先アクセス

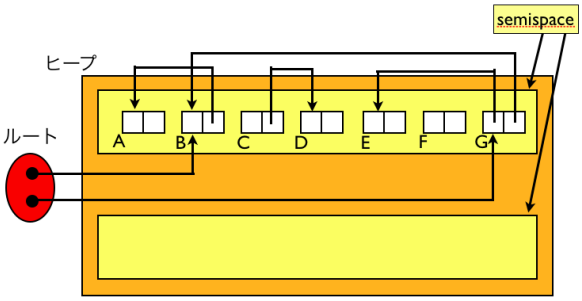

図 3.11　幅優先順によるコピー法 (コピー前)

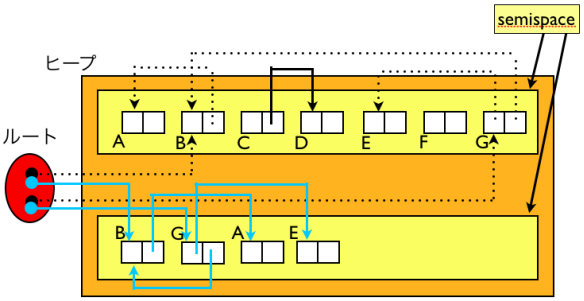

図 3.12　幅優先順によるコピー法 (コピー後)

3.4　階層的グループ化コピー法

このコピー法 (文献 [5]) は，メモリアクセスの空間的局所性を向上させるために提案されたコピー法である (図 3.13)．3.3 節の幅優先コピー法または深さ優先コピー法を利用して，ヒープ内のデータ構造を木の深さごとにグループをつくる．

3.5　選択スイープ法

マークスイープ法はヒープサイズが大きい場合，ガベージコレクションの回数が減少することで，1 回のガベージコレクションでごみを大量に回収可能である．しかし，1 回のガベージコレクション時間計算量が増加する．そのときの時間計算量は，ヒープ内で使用するメモリの量 (生きているオブジェクトの総数 A) に比例するため，$O(A)$ となる．

その後のスイープフェーズはごみを回収する．その回収にはフリーリストと呼ばれるリストに記憶させる．スイープフェーズの時間計算量は，ヒープサイズ H に比例した時間

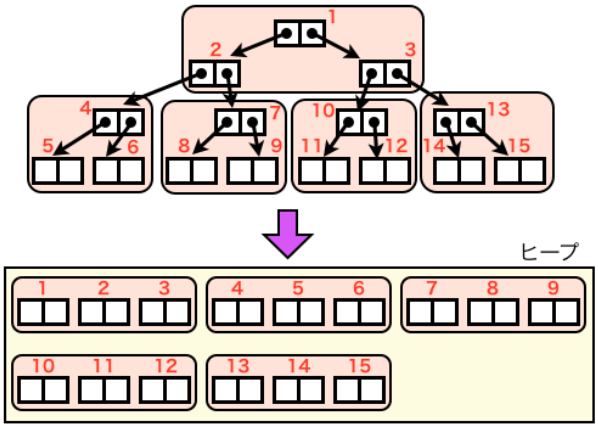

図 3.13　階層的コピー法

がかかる．つまり，$O(H)$ となる．

文献 [9] の研究では選択スイープ法 (Selective sweeping) を提案し，スイープフェーズの時間計算量がヒープサイズに比例しない手法について紹介している．

マークフェーズ (図 3.14) ではマークスイープ法と同様に行なう．ただし，生きているオブジェクトのアドレスを記憶する．そのために，ヒープ領域のアドレスは図 3.14 のように仮定する．また，生きているオブジェクトのアドレスを記憶するための配列を用意する必要がある．まず，ルートからオブジェクトを辿ると，最初に 5 番地，次に 5 番地の左スロットから辿って 0 番地，次に 1 番地，最後に 7 番地の順に配列に記憶していく．すると，配列に記憶した要素の番地は 5，0，1，7 と並んでいる．その時間計算量は，生きているオブジェクトの総数を参照するため，$O(A)$ がかかる．

スイープフェーズ (図 3.15) では，基数ソートとアドレスのスキャンを行なう．先ほどの配列要素を基数ソートで降順に並べ替えると，0，1，5，7 となる．これにより生きている 2 つのオブジェクトの番地の差分を取得し，ごみが存在する番地を確認できる．図 3.15 のように 1 番地と 5 番地の隙間には 3 つのごみの固まりがあることが分かる．そして，5 番地と 7 番地の隙間には 1 つのごみがある．フリーリストはそれらのごみをまとめて記憶させる．つまり，図 3.15 のように矢印がある番地をスキャンすれば，容易にごみを回収できる．そのための時間計算量は，配列要素数，つまり生きているオブジェクトのアドレス数に比例するため，$O(A)$ となる．スイープフェーズの時間計算量は，基数ソートとスキャン時間がかかるため，$O(A) + O(A)$ となる．

しかし，この手法は生きているオブジェクトが多いほど，オーバーヘッドが大きくなってしまう．そこで，マークスイープ法と選択スイープ法の切り替えを行なう適応スイープ法を提案している．切り替え方法は前回のガベージコレクション中に，生きているオブジェクトの検出数に基づいて，その数の閾値によって行なわれる．

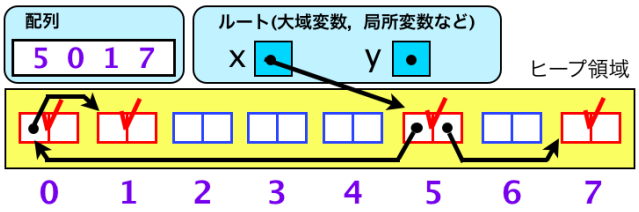

図 3.14　選択スイープ法のマークフェーズ

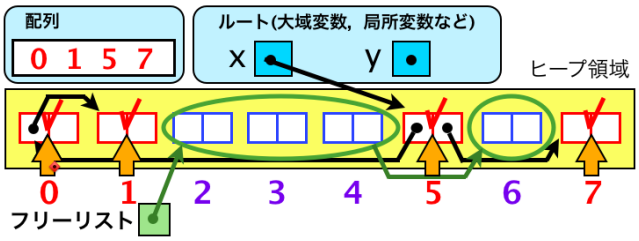

図 3.15　選択スイープ法のスイープフェーズ

第 4 章

提案手法

4.1 参照の局所性の利用

ミューテータによってオブジェクトが生成される．それらのオブジェクトがヒープ上で長く生き続けることはあまりない．ほとんどのオブジェクトがごみになってしまう．つまり，ヒープ上では短寿命のオブジェクトが多く占め，長寿命のオブジェクトが少ない．しかし，長寿命のオブジェクトは繰り返しアクセスされる線形リストや木構造のデータ構造を持つことで，連続してアクセスされることが多い．それらのデータ構造は親オブジェクトから子オブジェクトへと辿っていき，データをアクセスする．従って，新しいオブジェクトを生成し，新しい要素として追加する．時には長寿命のオブジェクトは親子関係のオブジェクトの配置に気を配る必要がある．しかし，マークスイープ法や遅延スイープ法では親子関係のオブジェクトが同じラインに配置されないことが多い．そのため，キャッシュミスが起き，空間的局所性が悪くなる．それにより，アクセス時間が長くなり，処理速度が遅くなる．

長寿命のオブジェクトは親子関係のオブジェクトを連続してアクセスする．これは多くのプログラムで見られている (文献 [12] を参考)．このようなオブジェクトは「参照の局所性 (2.1.1 節)」を考慮する必要がある．親子関係のあるオブジェクト群をヒープ領域上で互いに近い位置に配置しておくことで，利用したいデータが同じキャッシュライン上に読み込まれる可能性が高くなる．それにより，ミューテータが高速に動作することができる．

一般的にミューテータが高速に処理するには，図 4.1 のように，同じ色のついたオブジェクト群を同じラインに配置するように管理することで，親子関係のあるオブジェクトが同じキャッシュラインに入るようになり，空間的局所性が向上できる．図 4.2 は，図 4.1 をヒープ領域上で表したものである．しかし，実際は親子関係のオブジェクト間に関係のないオブジェクトが生成されてしまい，空間的局所性が悪くなる．図 4.3 のように色のついたライン単位でオブジェクト群を管理できない．

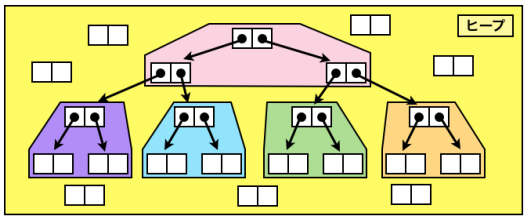

図 4.1　親子関係のあるデータの例

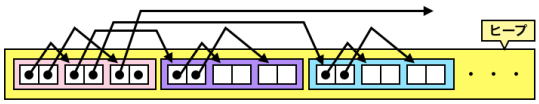

図 4.2　効率の良い配置の例

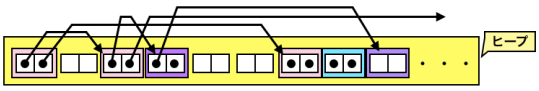

図 4.3　効率の悪い配置の例 (キャッシュミスを起こしやすい配置)

4.2　遅延スイープ法におけるアロケーションの問題点

2.4 節で説明したように，遅延スイープ法はスイープポインタを用いることで，スイープとアロケーションを同時に行なう．しかし，親子関係のあるオブジェクト群がヒープ領域上で互いに近い位置に配置されない．図 4.4–4.7 で示すように，ヒープのアドレス順にごみを探すため，親子関係のオブジェクトを考慮しないで配置を行なう．その結果，ミューテータが親子関係のあるオブジェクトを連続して参照するときに，それらのオブジェクトが離ればなれになっているため，処理速度が遅くなり，アクセス時間が長くなる．

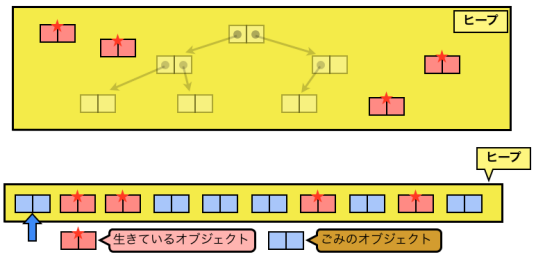

図 4.4　遅延スイープ法のアロケーション処理過程 1

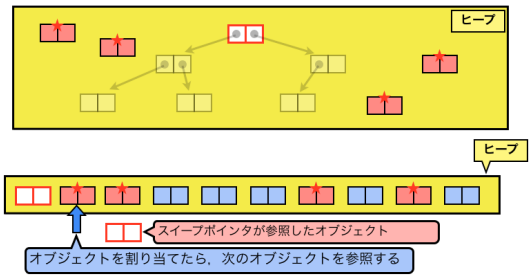

図 4.5　遅延スイープ法のアロケーション処理過程 2

遅延スイープ法の局所性を改善するには，長寿命のオブジェクトがヒープ領域上で互いに近い位置に配置することが必要である．その方法としては，ヒープ領域上のある領域部分を長寿命のオブジェクトしか配置できないようにすることである．短寿命のオブジェク

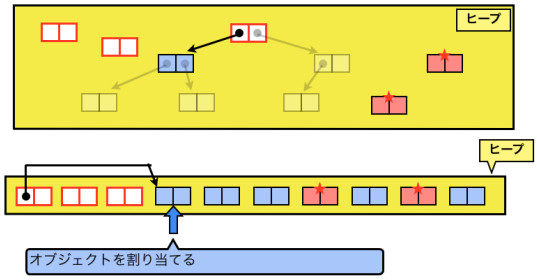

図 4.6　遅延スイープ法のアロケーション処理過程 3

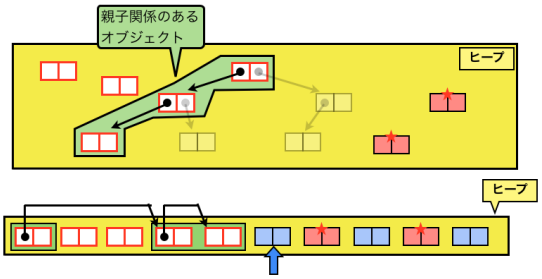

図 4.7　遅延スイープ法のアロケーション処理過程 4

トはその領域に配置されないようにする．一般的にヒープ領域上には多くのごみが存在する．ある量のごみの領域をチャンクとして管理し，長寿のオブジェクトのために使うようにする (図 4.8–4.9)．そして，新しく子のオブジェクトを生成する際には，その親となるオブジェクトが属しているチャンクから割り当てるようにする．そのためには，親オブジェクトが属しているチャンクを特定する必要がある．

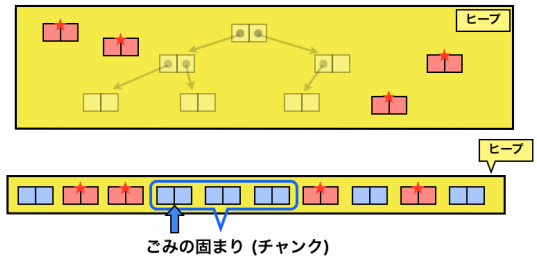

図 4.8　ヒープ領域からチャンクを探す処理過程

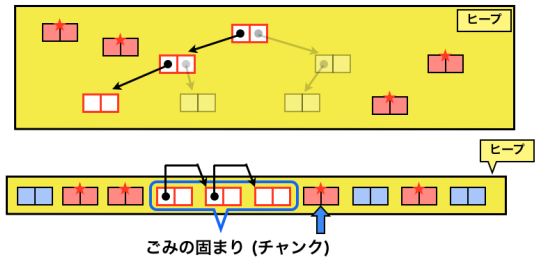

図 4.9　親子関係のオブジェクトがチャンクによって配置される処理過程

4.3　提案手法の概要

長寿命のオブジェクトに対して新しくオブジェクトを生成して追加する際には，ヒープ領域上で互いに近い位置に配置するために，親オブジェクトが属しているチャンクから割り当てを行なう必要がある．しかし，一般のプログラミング言語処理系ではオブジェクトを生成するとき，長寿命か短寿命のオブジェクトかどうかの判断をせずに，オブジェクトを使用する．つまり，長寿命と短寿命のオブジェクトの配置関係を考えずに配置してしまっている．長寿命である親子関係のオブジェクトの配置が離ればれになり，局所性が悪くなる．局所性を向上するには，それらを判断して配置する必要がある．

関連研究の文献 [4] ではプログラムコードを解析して，親子関係のオブジェクトはヒー

プ領域上の近辺に配置するようにしている．そのためのヒントはプログラムコードの構造から知ることができる．従って，プログラマがその構造を知っていて，親子関係のオブジェクトが分かっていて，そのオブジェクトだけをあるヒープ領域区分に配置できれば，空間的局所性を向上できる．

そこで空間的局所性を向上させる方法は，プログラマが親子関係のあるオブジェクトだと判断した場合，そのオブジェクトがどこの親に属しているかを判定し，親子関係を同じキャッシュラインに配置できるようにする．つまり連続して参照されるオブジェクトをあるヒープ領域区分だけに配置できるようにする．それには，まずプログラマが関係のあるオブジェクトだと分かっているときに，プログラムコードにその関係を判別するコードを追加する．メモリ領域からラインサイズの専用チャンクを割り当てておき，オブジェクトに子オブジェクトを追加するときに，親となるオブジェクトが属する専用チャンクから割り当てを行なうようにする．

具体的には以下のとおりである．

ソースコード 4.1 と図 4.10–4.11 は，高さ 2 の二分木を作成する際に，子と子の作成中にごみのオブジェクトを作成する Scheme のソースコードである．`define` は変数の宣言を示し，`tree` は変数である．`cons` は 2 つのアドレスを格納したオブジェクトを作成する．`cons` が呼ばれると，そのオブジェクトがメモリ領域に割り当てられる．`tree` には片方に 1 へのアドレスを，もう片方には 2 へのアドレスを入れる．1 へのアドレスが入っているほうを `car`，2 へのアドレスが入っているほうを `cdr` と呼ぶ．`(f x)` や `(g y)` は関数呼び出しであり，ここではごみや他のデータなどを作成する．`set-car!`は cons の car 部分を値やポインタに書き換える．`set-cdr!`も `set-car!`と同様に cdr 部分を書き換える．このような処理では親子関係のオブジェクトが離ればなれになり，空間的局所性が悪くなる．そこで，このコードを専用チャンクに割り当てるためのコードを追加する．

ソースコード 4.1　高さ 2 の二分木を作成するソースコード

```
1 (define (main)
2     (define tree (cons 1 2))
3     (f x)
4     (set-car! tree (cons 1 2))
5     (g y)
6     (set-cdr! tree (cons 3 4))
7   )
```

上記のコードをソースコード 4.2 のようにする．このソースコードは元々のコードから `cons` を `consA` に，`set-car!`を `set-car-A!`に，`set-cdr!`を `set-cdr-A!`に変更した．

このコードでは，他の処理によりごみや他のオブジェクトが生成されても，専用チャンクが割り当てられているため，`consA` で生成されたオブジェクトはそのチャンクに配置され，空間的局所性を高めることができる．`set-car-A!`や `set-cdr-A!`は最近アクセスした `consA` のオブジェクトのアドレスを記憶しておく．また，親オブジェクトのアドレス

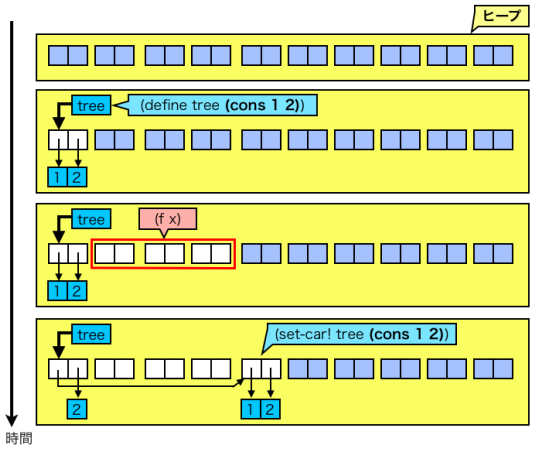

図 4.10　ソースコード 4.1 によるオブジェクト割り当ての処理過程 1

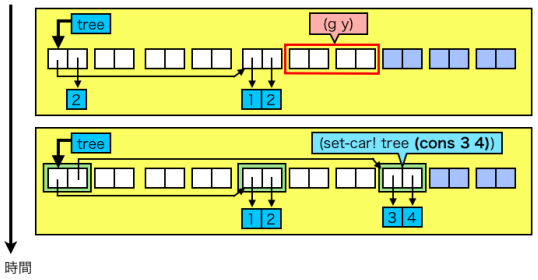

図 4.11　ソースコード 4.1 によるオブジェクト割り当ての処理過程 2

記憶はプログラムに依存するが，例えば，carA や cdrA などによってアクセスされたオブジェクトのアドレスを覚えておく．

ソースコード 4.2　専用チャンクからオブジェクトを割り当てるように修正したソースコード

```
1 (define (main)
2     (define tree (consA 1 2))
3     (f x)
4     (set-car-A! tree (consA 1 2))
5     (g y)
6     (set-cdr-A! tree (consA 3 4))
7   )
```

ソースコード 4.2 は図 4.12–4.13 のように処理される．図 4.12 では，consA のオブジェクトが始めて生成されるとき，consA 用のチャンクが存在しないため，チャンク (緑色で囲んでいるオブジェクト) のメモリ領域を割り当てる必要がある．consA が呼ばれるたびに，親となるオブジェクトが属するチャンクから未使用オブジェクト (緑色) が配置される．それ以外の通常の (短寿命の) オブジェクトの生成ではごみのある位置に配置する．親子関係のオブジェクトはヒープ領域上で互いに近い位置に配置するための専用のチャンクから割り当てを行なう．

また，複数の親子関係に対応できるようにチャンクも複数用意する必要がある．それを図 4.14–4.17 で説明する．子オブジェクトを生成する際に，どのチャンクに親オブジェクトが属するかを判断できるようにオブジェクトの追加の直前にアクセスした (赤い四角を最後に辿った) オブジェクトを親オブジェクトと仮定し，そのアドレスを記憶しておく (図中の矢印)．図 4.15 で示すようにオブジェクトが根を伸ばす (子を作る) 場合には，親オブジェクトが属するチャンクに未使用オブジェクトがある場合にはそのオブジェクトを割り当てるようにする．

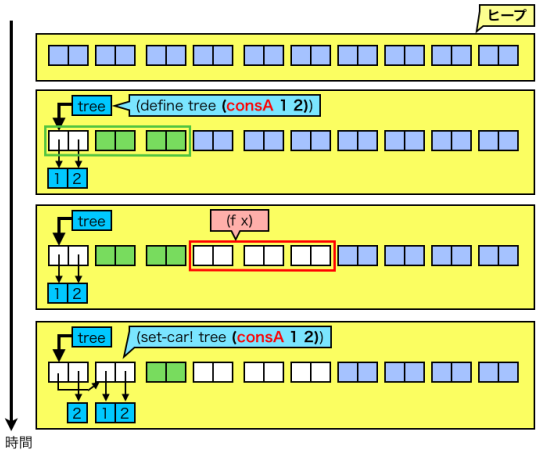

図 4.12　ソースコード 4.2 によるオブジェクト割り当ての処理過程 1

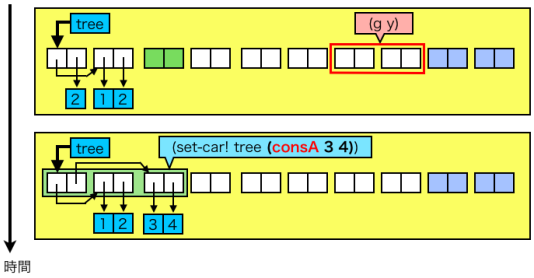

図 4.13　ソースコード 4.2 によるオブジェクト割り当ての処理過程 2

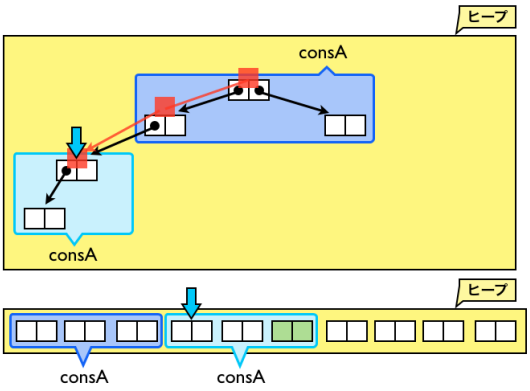

図 4.14　木構造のオブジェクト割り当ての処理過程 1

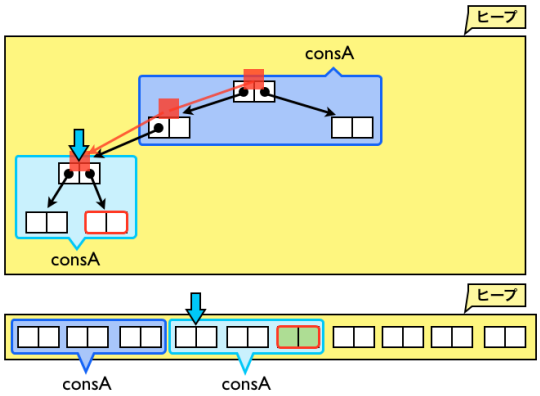

図 4.15　木構造のオブジェクト割り当ての処理過程 2

もし図 4.16 で示すように consA の根を伸ばす際にチャンクが残っていない場合には，図 4.17 で示すように，新たなチャンクを割り当てる．

そして，木構造データが独立して複数ある場合は，図 4.18–4.19 のように consB を用意して，親が属するオブジェクトの近くに配置できるようにする．

提案手法は，子オブジェクトの生成の際に親オブジェクトを判断できるため，親子のオブジェクトをヒープ領域上で互いに近い位置に配置することができる．

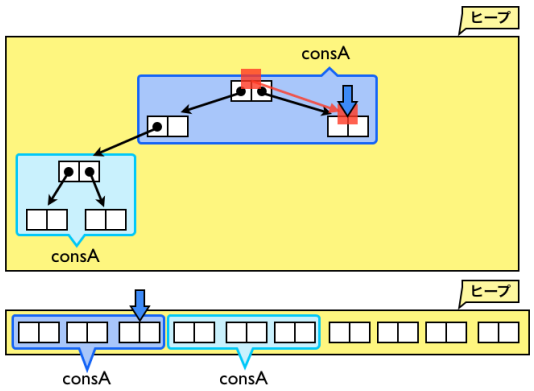

図 4.16　木構造のオブジェクト割り当ての処理過程 3

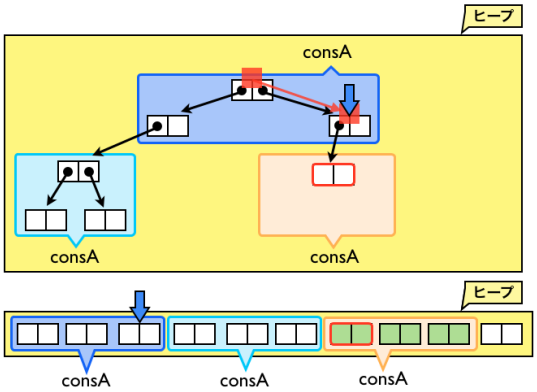

図 4.17　木構造のオブジェクト割り当ての処理過程 4

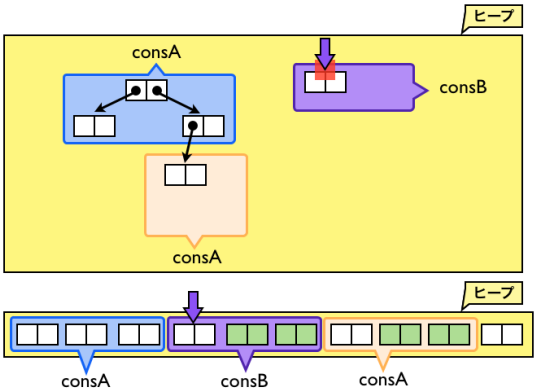

図 4.18　複数の親子関係がある場合のオブジェクト割り当ての処理過程 1

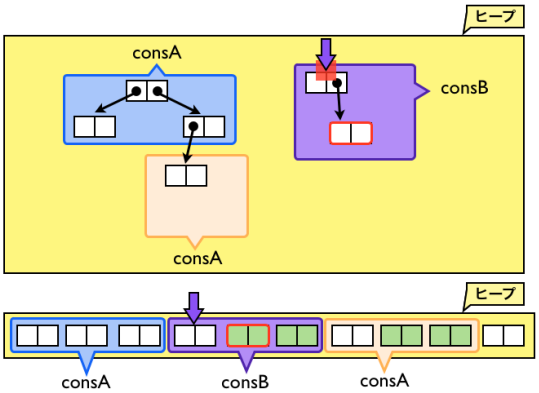

図 4.19　複数の親子関係がある場合のオブジェクト割り当ての処理過程 2

第 5 章

設計

5.1 親オブジェクトが属するチャンクの特定

今回の手法では，親子関係にあるオブジェクトをヒープ領域上で互いに近い位置に配置するために，親オブジェクトが属するチャンクの特定が必要である．また，チャンクがヒープ領域上のどこに存在しているか，チャンク内にいくつの未使用オブジェクトが存在しているかを管理する必要がある．ただし，それらの管理について注意しておくべきことがある．注意点は，その管理に伴うメモリ消費やメモリアクセスの効率である．そこで本研究では，チャンク管理用のメモリ領域をチャンク内の未使用オブジェクト領域にとることにした．チャンク外で管理してしまうと，メモリ領域上に新たな領域を使い，メモリの消費が多くなってしまう (図 5.1)．一方，チャンク内に管理領域を取る方法では，余分なメモリを消費せず，また，メモリアクセスの局所性の点でも都合が良い．

その注意点を考慮して以下で説明する．

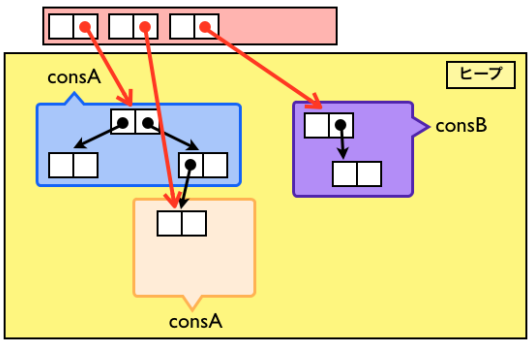

図 5.1 効率の悪いチャンク管理の例

5.2 ラインの境界

親オブジェクトが属するチャンクに，チャンクを特定できるようライン境界に合わせて配置する．チャンクはヒープを区分して管理する (図 5.2)．これによりチャンクとライン

の境界が一致するので，ビット演算を使用してチャンクの先頭を判別できるようにする．

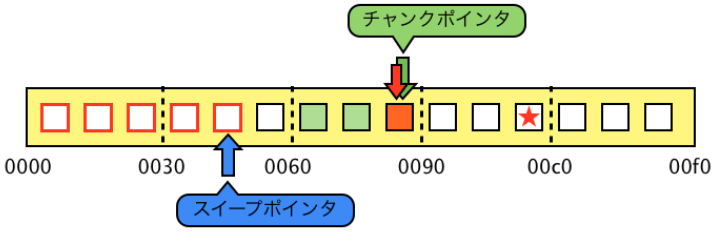

図 5.2　ラインサイズの管理

5.3　チャンクの管理方法

チャンクの管理には図 5.3–5.5 のように行なう．始めて親子関係のオブジェクトを割り当てる際，新しいチャンクをライン境界上に割り当てる．チャンクの先頭の未使用オブジェクトの car 部にはこのラインがチャンクとして利用されることを識別できるように特別なタグ (図中の水色の箱) を目印として入れておき，チャンクの先頭のオブジェクトの cdr 部にはチャンク内の未使用オブジェクトを指すようにする (図 5.3)．子オブジェクトは cdr 部が指しているオブジェクトから割り当てていく．割当て後は，cdr 部を 1 つ左のオブジェクトを指すように更新する．親子関係のオブジェクトを隣同士にするには，ミューテータが長寿命のオブジェクトを参照して，最後に辿ったオブジェクトを覚えておく必要がある．そのオブジェクトはチャンクから既に割り当てているオブジェクト (図中の赤い矢印) である．既に割り当てているオブジェクトを記憶しておくことで，子オブジェクトを割り当てるときに同じチャンク内に割り当てできるようにする (図 5.4)．そして，チャンク内の未使用オブジェクトを割り当てるときに，チャンク先頭のオブジェクトは管理用のメモリ領域としての役割を終える (図 5.5)．これで連続して参照されるオブジェクトを同じラインに割り当てることができる．また，子オブジェクトを割り当てたいときに，既にチャンクのオブジェクトがない場合には，始めて親子関係のオブジェクトを割り当てる際と同様なことを行なう．

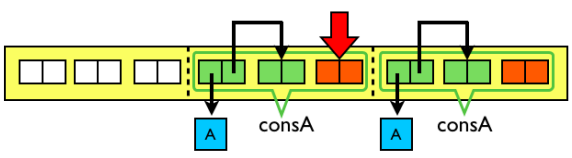

図 5.3　チャンク管理の処理過程 1

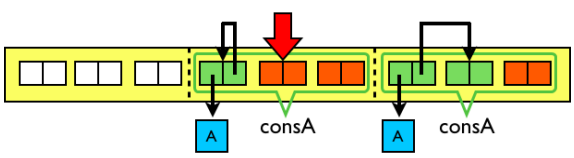

図 5.4　チャンク管理の処理過程 2

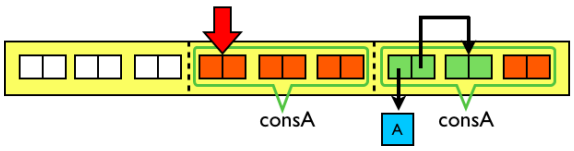

図 5.5　チャンク管理の処理過程 3

car 部にはタグ A と呼ばれる印をつけたことを説明した．これは次節で述べるようにチャンクポインタとスイープポインタの２つのポインタを使用して正確にごみやチャンクを探すために必要である．この印をもつオブジェクトで始まるラインサイズのヒープ領域は長寿命オブジェクトのために予約済みであることを示している．短寿命用オブジェクトのポインタであるスイープポインタがその割り当て予約済みの領域から短寿命のオブジェクトを割り当ててしまうと，親子オブジェクトの関係を破壊することになってしまう．それを防ぐためのものである．

また，スイープポインタ及びチャンクポインタはどちらも走査済みの生きたオブジェクトのマークをクリアする．従って，スイープポインタとチャンクポインタの間の領域は，一方のポインタによって走査済みであるが，他方のポインタによって再び走査される恐れがある．スイープポインタが先行する場合にチャンクポインタによる走査を再開する際は，次節で述べるようにスイープポインタの位置から再開するので再走査の心配はない．逆に，チャンクポインタが先行する場合にスイープポインタによる走査を再開する場合には，走査済みのチャンク (まだ未使用オブジェクトを持つライン) を効率よくスキップできるように，チャンク用マークテーブルを用意し，それがわかるようにした (図 5.6)．

スイープポインタが該当ラインにきたときに，チャンクが割り当て予約済みか割り当て済みかどうかを判断するようになる．

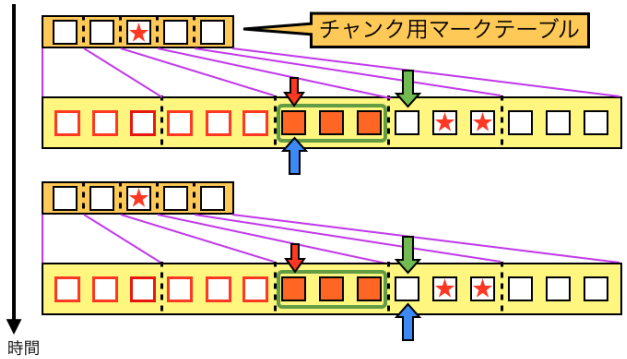

図 5.6　チャンク用マークテーブルの管理

5.4 チャンクポインタとスイープポインタ

一般のプログラミング言語処理系ではオブジェクトを生成するとき，長寿命か短寿命のオブジェクトかどうかの判断をせずに，オブジェクトを使用する．この判断をするために，今回は２つのポインタを使用する．長寿命のオブジェクトにはチャンクポインタを，短寿命 (通常) のオブジェクトには遅延スイープ法のスイープポインタを使用する．

5.4.1 チャンクポインタの処理

チャンクポインタの役割は，新たなチャンクの割り当てのためにヒープ領域上の空き領域をどこまで走査したかを覚えておくことである．

チャンクポインタの初期位置はヒープの先頭であり，そこからポインタを進め空き領域を走査する (図 5.7)．チャンクポインタはごみのチャンクを探しながら進める (図 5.8)．また，スイープポインタより低いアドレスを指している場合にはスイープポインタの位置より後の領域から探すようにする．つまり，スイープポインタが指す位置に移動する (図 5.9)．そして，チャンクの管理のために次のラインの境界を指す位置に移動する (図 5.10)．スイープポインタは短寿命のオブジェクトを割り当てているため，ヒープのアドレスの昇順に移動している．スイープポインタが通過したヒープ領域には，オブジェクトが既に配置しているため，チャンクポインタはその領域を参照する必要がない．

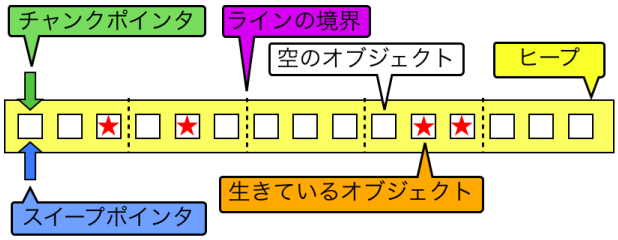

図 5.7　ポインタの初期位置

新たなチャンクの割り当て

始めて親子関係のオブジェクトを生成する際には，チャンクポインタがヒープを区分した境界の中からごみのチャンクを探す．この境界の中には生きているオブジェクトが 1 つもないと判断した場合にチャンクとして領域を確保する (図 5.11)．もし生きているオブジェクトが 1 つでも見つかった場合，チャンクポインタは次の境界からごみのチャンクを探す (図 5.12)．そのスキップしたラインは，スイープポインタによる走査においてもス

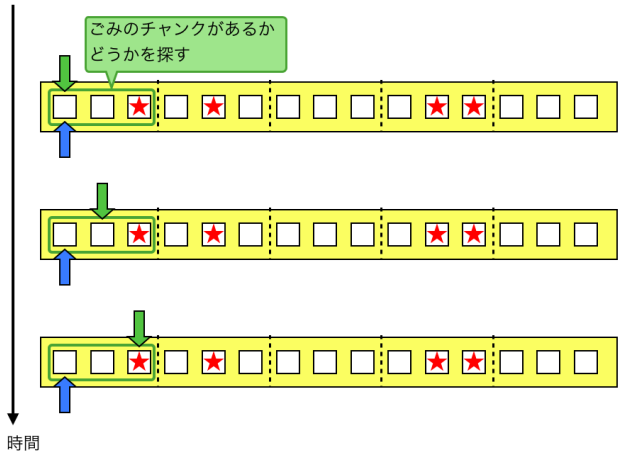

図 5.8　ごみのチャンクを探しながら進める処理過程

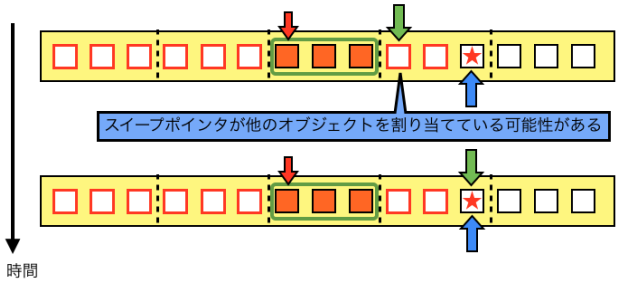

図 5.9　スイープポインタが指す位置に移動する処理過程

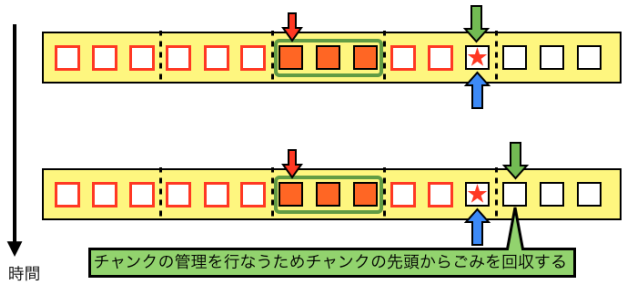

図 5.10　次のラインの境界を指す位置に移動する処理過程

キップする必要がある．そこで，チャンク用マークテーブルを用意して，スイープポインタが該当ラインにきて，そのマークがある場合にはスキップする．

　もし仮にチャンクポインタがヒープの終わりまできた場合には，マークフェーズを行なう．また，チャンクポインタがヒープの終わりに達したらマークテーブルをオールリセットすればよい．

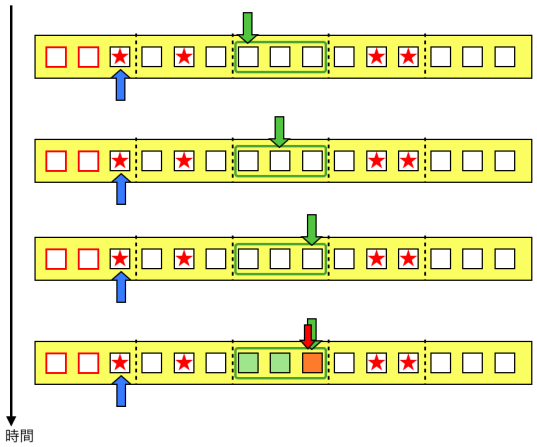

図 5.11　新たなチャンクの割り当ての処理過程

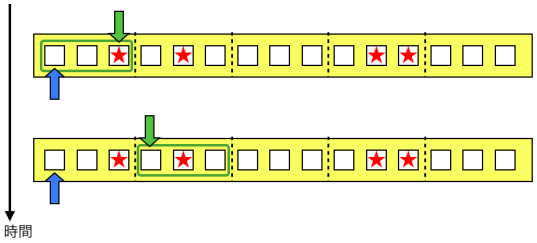

図 5.12　チャンクポインタが次のライン境界に移動する処理過程

親が属するチャンクのオブジェクトの割り当て

長寿命のオブジェクトが子オブジェクトを割り当てる際には，親が属しているチャンクから未使用オブジェクトを割り当てる．割り当てる際には，チャンクから割り当てたオブジェクトの近くにオブジェクトを割り当てる (図 5.13)．チャンクから未使用オブジェクトが割り当てできなくなったら，新たなチャンクを割り当てる．

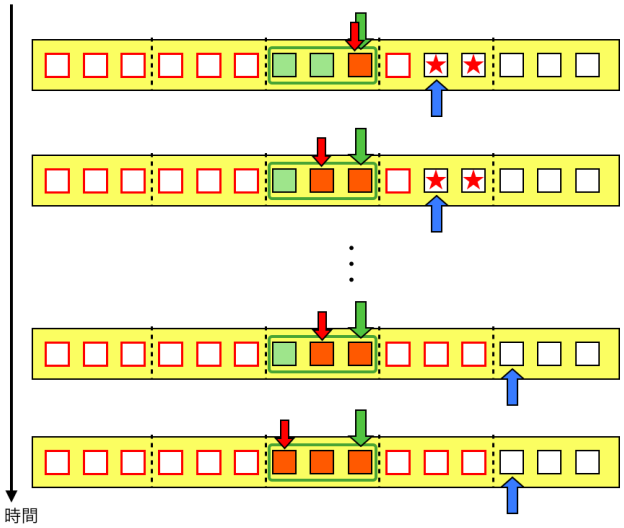

図 5.13　子オブジェクトの割り当ての処理過程

5.4.2 スイープポインタの処理

短寿命のオブジェクトにはスイープポインタを使用する．スイープポインタの役割は，マークスイープ法や遅延スイープ法のそれと同じである．しかし，チャンクによるオブジェクトを配慮する必要がある．チャンクポインタが既に割り当てられたところに新たなオブジェクトを配置してしまうと，親子関係を持つオブジェクトが連続してアクセスできなくなってしまう．それを配慮した方法を説明する．

スイープポインタの初期位置はチャンクポインタと同様にヒープの先頭からポインタを進め始める (図 5.7)．スイープポインタはヒープのアドレスの昇順に移動している．スイープポインタが生きているオブジェクトかごみのオブジェクトかを判断して，ごみを回収している．ごみの場合にはそれを回収して，新たなオブジェクトとして割り当てる．生きているオブジェクトの場合には，印があるのでその印を消す．これにより，スイープポインタが参照した領域にはごみがない．

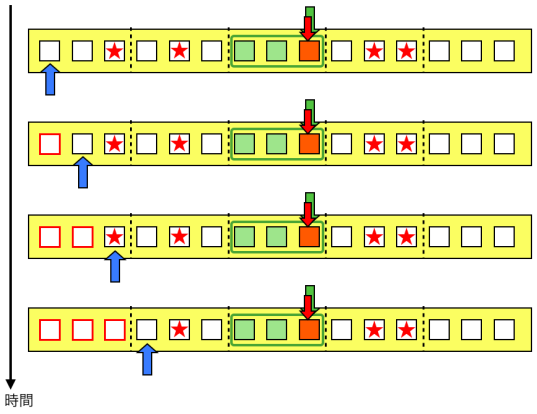

図 5.14　スイープポインタによるオブジェクト割り当ての処理過程

しかし，5.4.1 節で説明したように，チャンクポインタがあるため，未使用オブジェクトを持つチャンク (未使用オブジェクトを持たないチャンクはもはやチャンクとして認識されない) を判断しながらスイープを行なう．もしチャンク用マークテーブルを調べて，ラインがチャンク (未使用オブジェクトを持つライン) であることがわかれば，次の境界から探すようにする (図 5.15–5.16)．

スイープポインタがヒープの終わりまできた場合には，マークフェーズが動作する．マークフェーズ後は，チャンクポインタとスイープポインタがヒープの始めからチャンクを探していく (図 5.17)．

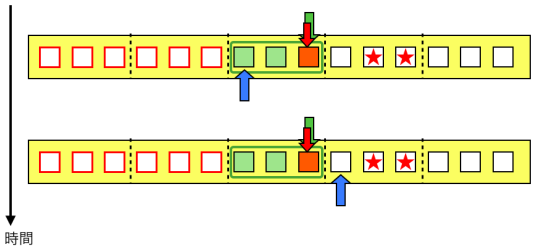

図 5.15　割り当て予約済みがある場合のスイープポインタの移動処理過程

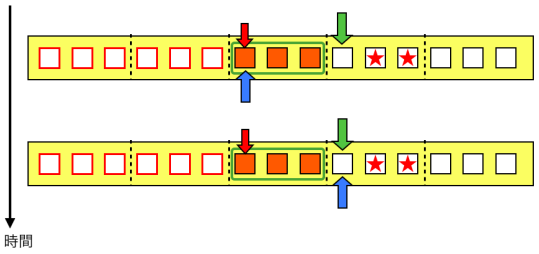

図 5.16　割り当て済みがある場合のスイープポインタの移動処理過程

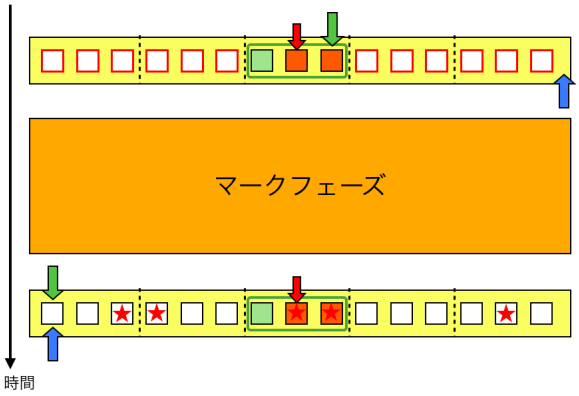

図 5.17　チャンクポインタとスイープポインタをヒープの始めに再配置する処理過程

5.5　アルゴリズム

ソースコード 5.1 は短寿命のオブジェクトの新しいオブジェクトを割り当てるときに呼出されるコードである．car 部には x，cdr 部には y を指すようにする．NEWCELL(z) の処理が呼び出されると，スイープポインタが動作する．

ソースコード 5.1　短寿命のオブジェクトを割り当てるコード

```
1 OBJP cons (OBJP x, OBJP y){
2     OBJP z;
3     NEWCELL(z);
4     CAR(z) = x;
5     CDR(z) = y;
6 }
```

ソースコード 5.2 は長寿命のオブジェクトの新しいオブジェクトを割り当てるときに呼出されるコードである．latest_access_A は割り当てた z のオブジェクト (図中の赤矢印) を記憶するための変数である．次回 consA が呼出される際，この変数が指すオブジェクトを親オブジェクトとみなして適切なチャンクを選ぶ．MYMETHOD_NEWCELL(z) の処理

が呼び出されると，チャンクポインタが動作する．

ソースコード 5.2　長寿命のオブジェクトを割り当てるコード

```
1 OBJP consA (OBJP x, OBJP y){
2     OBJP z;
3     MYMETHOD_NEWCELL(z);
4     latest_access_A = z;
5     CAR(z) = x;
6     CDR(z) = y;
7 }
```

また，親オブジェクトのアドレス記憶はプログラムに依存するため，consA の他に carA や cdrA などによってアクセスされたオブジェクトを覚えておく (ソースコード 5.3)．ただし，どこで latest_access_A を更新すべきかは実行するプログラムに依存するので，プログラムに合わせて適切な仕様の cons や car 等の関数を用意する必要がある．

ソースコード 5.3　アクセスされたオブジェクトを覚えるコード

```
1  OBJP carA (OBJP x){
2      ....
3      latest_access_A = x;
4      ....
5  }
6
7  OBJP cdrA (OBJP x){
8      ....
9      latest_access_A = x;
10     ....
11 }
```

5.2 節より各チャンクの先頭のアドレスを得るには，ソースコード 5.4 に示すビット演算を利用して，求めるようにする．0xfff...c0 は 16 進数であり，例として 0x000...06 & 0xfff...c0 を行なうと，0x000...04 を返す．なぜ 0xfff...c0 に設定しているのかは，L1 キャッシュが 64Byte で格納できるため，そのチャンクの先頭を指すように設計する．先ほどの親オブジェクトを記憶する latest_access_A を使用して，chunk_adr(latest_access_A) のようにすることで親オブジェクトが属するチャンクの先頭を指すことができる．また，親オブジェクトが属するチャンクから未使用オブジェクトを割り当てるには，cdr(chunk_adr(latest_access_A)) で親オブジェクトが属するチャンクの先頭から辿って cdr 部が指しているオブジェクトを割り当てる．

ソースコード 5.4　チャンクの先頭を指すビット演算のコード

```
1 OBJP chunk_adr (OBJP x){
2     return x & 0xfff...c0;
3 }
```

あるオブジェクトがチャンクの先頭かどうかを確認するコードをソースコード 5.5 に示す．例としては，is_top_of_chunkA(chunk_pointer) で確認し，先頭の場合はそこからごみを集めて回収し，チャンクを割り当てる．また，スイープポインタ (sweep_pointer)

が割り当て中のチャンクかを判断できるために，is_top_of_chunkA(sweep_pointer) で確認する．先頭なら割り当て予約済みや割り当て済みがあるかどうかを確認する処理に進める．

ソースコード 5.5　各ラインの先頭かどうかを確認するコード

```
1 OBJP is_top_of_chunkA (OBJP x){
2     return chunk_adr(x) == x;
3 }
```

ソースコード 5.6 は次のチャンクの先頭に移動するための処理である．

ソースコード 5.6　次のチャンクの先頭に移動するコード

```
1 OBJP chunk_move (OBJP x){
2     return x + CACHE_LINE_SIZE;
3 }
```

あるラインがチャンクかどうかを判定するコードをソースコード 5.7 に示す．

ソースコード 5.7　割り当て予約済みがあるかを確認するコード

```
1 OBJP is_chunkA (OBJP chunk){
2     return CAR(chunk) == 'A';
3 }
```

チャンクポインタのアルゴリズム

5.4.1 節によるチャンクポインタのアルゴリズムをソースコード 5.8 と 5.9 に示す．

ソースコード 5.8 は MYMETHOD_NEWCELL が呼ばれたときに処理を行なう．is_chunk_A(chunk_adr(latest_access_A)) は，親が属するチャンクに割り当て予約があるかどうかを確認する．確認ができたなら，そのチャンクから未使用オブジェクトを割り当てる．確認ができないなら，allocate_chunk() より新たなチャンクを割り当てる．チャンクポインタがヒープの終わりまで走査し，新たなチャンクを割り当てできない場合には，スイープポインタが生きているオブジェクトの印を消していき，マークフェーズ (mark_phase()) を始める．マークフェーズ後は，チャンクポインタとスイープポインタをヒープの始めに配置し，allocate_chunk() で新たなチャンクを割り当てる．それでも新たなチャンクを割り当てできない場合には，エラーメッセージを表示して処理を停止する．

ソースコード 5.8　チャンクポインタによるオブジェクトの割り当て処理

```
1 OBJP _MYMETHOD_NEWCELL()
2 {
3     // 親が属するチャンクに割り当て予約がある場合
4     if (is_chunk_A(chunk_adr(latest_access_A))){
5         new_cell = CDR(chunk_adr(latest_access_A));
6         --CDR(chunk_adr(latest_access_A));
7         return new_cell;
```

```
8       }
9       // 親が属するチャンクに割り当て予約がない場合
10      else{
11          // 新たなチャンクが割り当てる
12          new_cell = allocate_chunk();
13
14          // 新たなチャンクが割り当てできるなら
15          if(new_cell != NULL){
16              return new_cell;
17          }
18          // 新たなチャンクが割り当てできないなら
19          else{
20              // 生きているオブジェクトの印を消す
21              for(; sweep_pointer < heap_end; sweep_pointer++){
22                  sweep_pointer.mark_bit = NO_MARK;
23              }
24
25              // マークフェーズを動作する
26              mark_phase();
27
28              // チャンクポインタとスイープポインタをヒープの始めに配置する
29              sweep_pointer = heap_org;
30              chunk_pointer = heap_org;
31
32              // 新たなチャンクが割り当てる
33              new_cell = allocate_chunk();
34
35              // 新たなチャンクが割り当てできないなら
36              if(new_cell == NULL){
37                  error();
38              }
39          }
40      }
41  }
```

5.4.1 節による新たなチャンクを割り当てるコードをソースコード 5.9 に示す．チャンクポインタがスイープポインタより低いアドレスを指している場合には，スイープポインタが指す位置に移動する．ごみを探す前に，ライン境界のアドレスに揃える必要がある．for 文でチャンクポインタ (chunk_pointer) でごみを探すときに，ごみが見つかった場合はごみの数 (garbage) を 1 つ増やす．生きているオブジェクトが見つかった場合は，ごみの数を 0 にし，次のライン境界の先頭を指すようにする．ごみのチャンクを集めることができた場合は，新しいオブジェクト new_cell には dtr が指すオブジェクトを割り当て，CAR(ptr) には割り当て予約済みを示す A を追加し，CDR(ptr) にはチャンク内のごみ (未使用オブジェクト) を指しておく．

ソースコード 5.9　新たなチャンクを割り当てる処理

```
1   OBJP allocate_chunk()
2   {
3       // ごみの数
4       garbage = 0;
5       // ライン境界の始め (ptr) と終わり (dtr) を指す
6       OBJP ptr, dtr;
7
8       // チャンクポインタがスイープポインタより低いアドレスを指している場合
9       if ( chunk_pointer < sweep_pointer ){
10          chunk_pointer = sweep_pointer;
11      }
12
```

```
13      // ライン境界のアドレスに揃える
14      if ( chunk_adr(chunk_pointer) < chunk_pointer &&
15           !is_top_of_chunk_A(chunk_pointer) )
16      {
17          chunk_pointer = chunk_move(chunk_adr(chunk_pointer));
18          ptr = chunk_pointer;
19      }
20
21      // ごみのチャンクを探す
22      for( ; chunk_pointer < heap_end; chunk_pointer++ ){
23          // ごみが見つかった場合
24          if( chunk_pointer.markbit == NO_MARK ){
25              garbage++;
26              dtr = chunk_pointer;
27
28              // ごみのチャンクを集めることができた場合
29              if( garbage == 4){
30                  new_cell = dtr;
31                  dtr--;
32                  CAR(ptr) = 'A';
33                  CDR(ptr) = dtr;
34                  return new_cell;
35              }
36          }
37          // 生きているオブジェクトが見つかった場合
38          else{
39              garbage = 0;
40              chunk_pointer = chunk_move( chunk_adr(chunk_pointer));
41          }
42      }
43
44      // ごみのチャンクがない場合には NULL を返す
45      return NULL;
46  }
```

スイープポインタのアルゴリズム

5.4.2 節によるスイープポインタの処理をソースコード 5.10 に示す．一般的には遅延スイープ法と同じである．しかし，チャンクポインタがあるため考慮しながら，スイープポインタを進める必要がある．チャンクポインタよりスイープポインタが先にヒープの終わりまで走査したら，マークフェーズを行い，チャンクポインタとスイープポインタはヒープの始めに配置する．そして，スイープポインタがごみを探すが，ごみが見つからなかった場合にはエラーメッセージを表示して処理を停止する．

ソースコード 5.10　スイープポインタによるオブジェクトの割り当て処理

```
1   OBJP _NEWCELL()
2   {
3       // ごみを探す
4       new_cell = sweeping_for_gc_lazysweep();
5
6       // ごみが見つかった場合
7       if (new_cell != NULL){
8           return new_cell;
9       }
10      // ごみが見つからなかった場合
11      else{
12          // マークフェーズを動作
13          mark_phase();
14
```

```
15          // チャンクポインタとスイープポインタをヒープの始めに配置する
16          sweep_pointer = heap_org;
17          chunk_pointer = heap_org;
18
19          // ごみを探す
20          new_cell = sweeping_for_gc_lazysweep();
21
22          // ごみが見つからないなら
23          if(new_cell == NULL){
24              error();
25          }
26      }
27  }
```

ごみの回収を行なうコードをソースコード 5.11 に示す．chunk_allocation_check は，スイープポインタがラインの先頭を指している場合，is_chunk_A より割り当て予約済みがあるかどうかを確認する．また，chunk_alloc_bit(チャンク用マークテーブル) より割り当て済みがあるかどうかを確認する．ある場合には次のラインの先頭に移動する．chunk_al_bit_adr はチャンク用マークテーブルの配列要素を求めるために使用する．ごみが見つからなかった場合には NULL を返す．

ソースコード 5.11　スイープポインタによるごみの回収処理

```
1   // chunk_alloc_bit の配列の要素を求める
2   #define chunk_al_bit_adr(OBJP x) (((x)-(heap_org))/(CACHE_LINE_SIZE))
3
4   OBJP sweeping_for_gc_lazysweep()
5   {
6       // ごみを探す
7       for (; sweep_pointer < heap_end; sweep_pointer++) {
8
9   // 各ラインの先頭を指しているかどうかを確認する
10  chunk_allocation_check:
11          // ラインの境界の先頭にいる場合
12          if( is_top_of_chunk_A(sweep_pointer) ){
13              // ラインの境界の先頭に割り当て予約済みがあるなら,
14              // 次のラインの先頭に移動する
15              if ( is_chunk_A(sweep_pointer) ){
16                  sweep_pointer = chunk_move(sweep_pointer);
17                  goto chunk_allocation_check;
18              }
19              // 割り当て済みがあるなら，次のラインの先頭に移動する
20              if( chunk_alloc_bit[ chunk_al_bit_adr(sweep_pointer) ]
21                      == MARK ) {
22                  sweep_pointer = chunk_move(sweep_pointer);
23                  goto chunk_allocation_check;
24              }
25          }
26
27          // ごみが見つかった場合
28          if (sweep_pointer.markbit == NO_MARK){
29              new_cell = sweep_pointer;
30              sweep_pointer++;
31              return new_cell;
32          }
33      }
34
35      // ごみが見つからなかった場合
36      return NULL;
37  }
```

第 6 章

実装

6.1 Scheme 言語処理系 SCM

SCM[2] は，インタプリタとコンパイラの両方を持つ Scheme 言語処理系である．C 言語で実装されており，Mac OS や Windows，Unix などのオペレーティングシステムで動作する．SCM のコンパイラ処理には「Scheme 言語で書かれたファイル」を「C 言語で書かれたファイル」に変換することができる．SCM の GC はマークスイープ法が実装されている．

6.1.1 オブジェクトの構成

オブジェクトは `SCM` 型の `car` と `cdr` の 2 ワードで構成されている．cdr 部の最下位ビットをマークビットとして使用している (文献 [3] を参考)．

6.1.2 マークテーブル

前節で説明したように，SCM では cdr 部の一部をマークビットとして使用するため，GC 中の cdr 部のポインタ値は正しい値ではない．そのため，GC をしながらプログラムの実行が進む遅延スイープ法や本提案手法を SCM に取り入れるには改造が必要となる．

そこで今回はマークテーブルを使用する (図 6.1)．これを使用することで，生きているオブジェクトの印の有無だけを確認することができる．例えば，スイープポインタでごみを回収するときに，生きているオブジェクトが見つかった場合にはマークテーブルが対応する印を消すだけで済む．

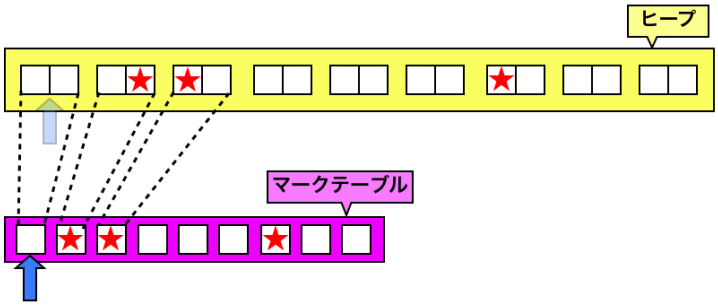

図 6.1　マークテーブルの管理

6.2　提案手法の実装

SCM の GC はマークスイープ法を行なっている．まず，マークスイープ法を遅延スイープ法に改造した．それから，5.5 節で説明した遅延スイープ法と提案手法のアルゴリズムを実装した．また，ヒープサイズは 1GByte に，オブジェクトサイズは 16Byte に，そしてチャンクサイズまたはラインサイズは 64Byte に設定した．

第 7 章

実験と考察

今回の実験は，遅延スイープ法と提案手法の性能差を評価する．

7.1 ベンチマーク

ベンチマークは，キャッシュミスを起こすベンチマークとキャッシュヒットを起こすベンチマークを作成し，使用する．つまり，生きているオブジェクト間にごみがあるものとないものを使用する．キャッシュミスを起こすベンチマークは，生きているオブジェクトの間に 3 つのごみを作成する．また，長寿命のオブジェクトは線形リストと二分木を使用する．評価は，作成した長寿命のオブジェクトを 10 回繰り返し参照した時間を測定する．線形リストは 100 万個の生きているオブジェクトを作成する．そして，二分木は深さ優先で作成し，高さ 20 とする．現段階の実装で実験を行なうと，GC を 1 回以上起こすとバグが発生するため，今回は GC を起こさない状態で実験を行なった．

7.2 実験結果と考察

線形リストによる実験結果 (図 7.1) は，ごみがない状態による遅延スイープ法では 4 ミリ秒かかり，ごみがある状態による遅延スイープ法では 11 ミリ秒かかった．そして，ごみがある状態による提案手法では 19 ミリ秒かかった．

二分木による実験結果 (図 7.2) は，ごみがない状態による遅延スイープ法では 44 ミリ秒かかり，ごみがある状態による遅延スイープ法では 73 ミリ秒かかった．そして，ごみがある状態による提案手法では 72 ミリ秒かかった．

提案手法の実装に不備があるため，遅延スイープ法の性能のほうが良い結果になってしまった．本来なら，提案手法はごみがある状態による遅延スイープ法より処理が速いと考えている．

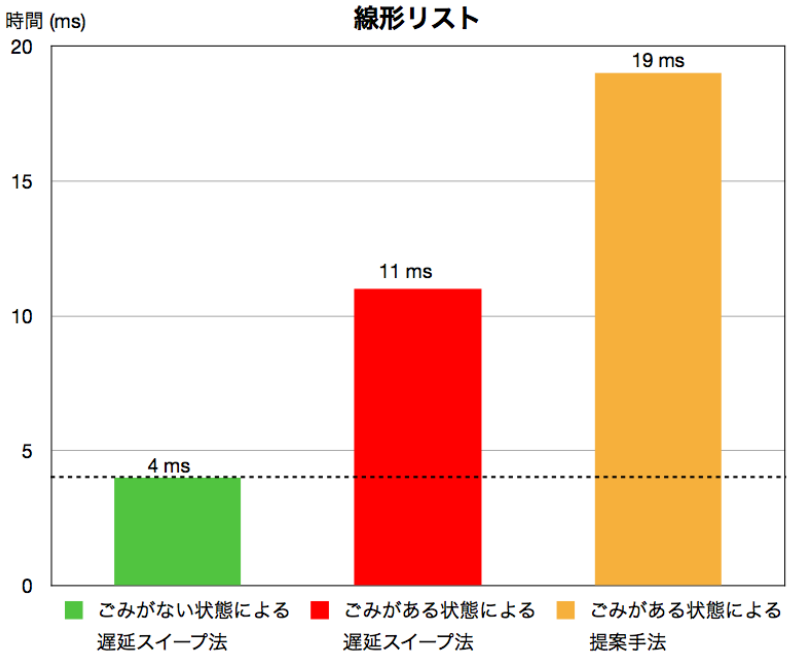

図 7.1　線形リストによる実験結果

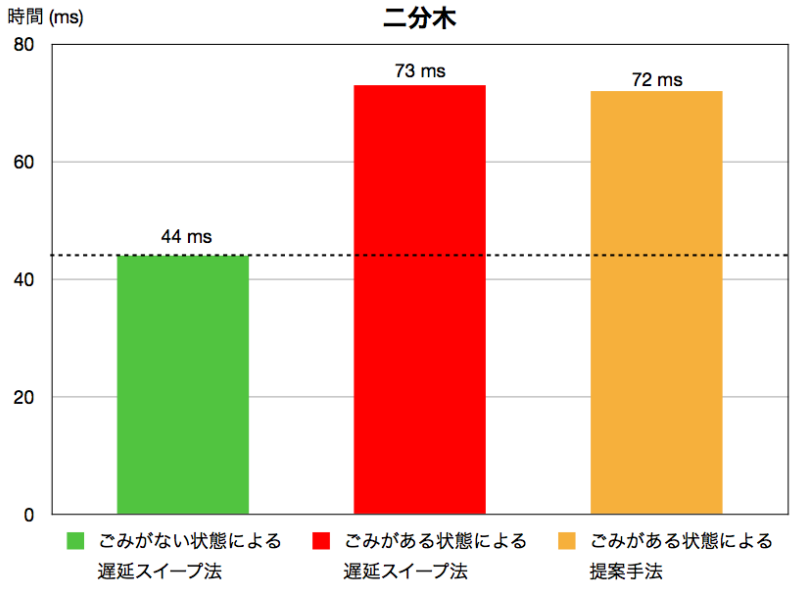

図 7.2　二分木による実験結果

第 8 章

まとめと今後の予定

遅延スイープ法はマークスイープ法の空間的局所性を改善できていないため，さらにメモリアクセスの局所性を向上させることが望まれる．キャッシュはライン単位で管理されるため，新たに割り当てるオブジェクトは最近参照したヒープ領域の近辺に配置できると良い．しかし，他のオブジェクトが既に割り当てられていて探すのが困難な可能性がある．本研究では親子関係のあるオブジェクトを生成し始めるときに，ごみのチャンクを割り当て，そこへ関係のあるオブジェクトを配置できるようにする手法を示した．その手法では，後に親子関係のあるオブジェクト群を生成する際は，専用のコンストラクタを明示的に用いるようにしている．このコンストラクタは親子関係のオブジェクトを参照の局所性に考慮してチャンクに配置する．

現段階では，実装を終えたところであり，キャッシュミスが起きるベンチマークを使用して実験を行なった．しかし，GC を 1 回以上起こすとバグが発生するため，今回は GC を起こさない状態で実験を行なった．結果は提案手法の実装に不備があるため，遅延スイープ法の性能のほうが良い結果になってしまった．提案手法が遅延スイープ法より高速に処理できると考え，実装し直す予定である．今後の予定はバグをなくし，様々なベンチマークを使用して提案手法の性能の向上率を調べる．

謝辞

本研究を遂行するにあたっては，いろいろな方々にお世話になりました．

まず，指導教員の小宮常康先生には日頃から熱心なご指導，そしてご鞭撻を賜わりました．また，ご多忙中にもかかわらず論文の草稿を丁寧に読んで下さり，大変貴重なご助言をいただきました．ここに厚く御礼申し上げます．

そして，本研究が行なえたことは，研究方針や方法論について議論をし，共に研究生活をおくってきた多田研，小宮研，鶴岡研，そして末田研の学生諸氏おかげでもあります．最後に，これらの皆さんに感謝いたします．

参考文献

[1] R.J.M.Hughes: “A semi-incremental garbage collection algorithm,” *Software – Practice and Experience*, Vol.12, No.11, pp.1081–1084, Nov.,1982.

[2] The SCM Implementation of Scheme: `http://people.csail.mit.edu/jaffer/SCM`, Aubrey Jaffer, Jan., 2013.

[3] 6.1.10 Data Type Representations: `http://people.csail.mit.edu/jaffer/scm/Data-Type-Representations.html#Data-Type-Representations`, Aubrey Jaffer, Jan., 2013.

[4] Hiroshi Inoue, Toshio Nakatani: “Identifying the Sources of Cache Misses in Java Programs Without Relying on Hardware Counters,” *ISMM' 12*, Jun., 2012.

[5] 八杉 昌宏, 小宮 常康, 伊藤 智一, 湯浅 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(PRO 21), pp.36–52, May., 2004.

[6] Hans-J. Boehm: “Reducing garbage collector cache misses,” *Proceedings of ISMM' 2000: ACM SIGPLAN International Symposium on Memory Management*, pp.59–64, Oct., 2000.

[7] Robert R.Fenichel and Jerome C.Yochelson: “A LISP garbage-collector for virtual-memory computer systems,” *Communications of the ACM*, Vol.12, No.11, pp.611–612, Nov., 1969.

[8] C. J. Cheney: “A Nonrecursive List Compacting Algorithm,” *Communication of the ACM*, Vol.13, No.11, pp.611–612, Nov., 1970.

[9] Yoo C.Chung, Soo-Mook Moon, Kemal Ebcioglu, and Dan Sahlin: “Selective sweeping,” *Software - Practice and Experience*, Vol.35, pp.15–26, Jan., 2005.

[10] 湯浅 太一: “汎用コンピューターでの実時間ごみ集め”, 日経サイエンス, pp. 56–71, Sep., 1988.

[11] 中村 成洋, 相川 光, 竹内 郁雄: “ガベージコレクションのアルゴリズムと実装”, 秀和システム, ISBN978-7980-2562-9, p.12–19, Jun., 2010.

[12] 中村 成洋, 相川 光, 竹内 郁雄: “ガベージコレクションのアルゴリズムと実装”, 秀和システム, ISBN978-7980-2562-9, pp.22, Jun., 2010.

[13] デイビット・A・パターソン, ジョン・L・ヘルシー: “コンピュータの構成と設計 第３版 [下]”, 日経 BP 社, ISBN978-4-8222-8267-7, pp.456–457, Jun., 2010.