スレッド別ごみ集めにおけるライトバリアの高速化

千 葉 雄 司 †

ごみ集めによる長い停止時間の発生頻度を軽減する方法に，スレッド別ごみ集めがある．スレッド別ごみ集めでは，ヒープ領域を分割して個々のスレッドに与え，スレッドは他のスレッドを止めずに単独でごみ集めを行う．スレッド単独でのごみ集めによる停止時間は，分割した領域の大きさに比例するので，ヒープ領域全体を一括してごみ集めする場合より短くなる．スレッド別ごみ集めを他のスレッドを止めずに実行できるのは，他のスレッドから参照できないデータ構造のみをごみ集めの対象とするからである．スレッド別ごみ集めの実装にあたっては，データ構造をスレッド別ごみ集めの対象になるものと，そうでないものに分類する必要がある．分類方法の 1 つである実行履歴を使う方法では，参照をクラス変数など複数のスレッドから参照可能な場所にストアしたか実行時にライトバリアを使って監視し，したならば参照が指示するデータ構造をスレッド別ごみ集めの対象から除外する．本論文ではライトバリアのオーバヘッドを最小限にとどめる方法について考察する．サーバ向けベンチマーク SPECjbb2000 を使って予備評価した結果，本論文で提案する方法では，ライトバリアから生じるオーバヘッドを 1%未満にできることが分かった．

High-performance Write Barriers for Thread-local Garbage Collection

Yuji Chiba†

Thread local garbage collection can lower frequency of long pause time for garbage collection because it divides the heap for each thread and the thread collects garbage in its own heap without interference with other threads. Pause time for thread local garbage collection is shorter than garbage collection for whole heap because thread local heap is smaller than whole heap. Thread local garbage collection does not interfere with other threads because it collects local data structures, which other threads cannot refer. An implementation of thread local garbage collection requires method to tell local data structures from global ones. One of the methods use write barriers to keep track of a reference if it is stored in non thread local objects, and then handle the data structure the reference points to as global. This paper presents an implementation of thread local garbage collection with minimum write barrier cost and shows result of preliminary evaluation on SPECjbb2000, benchmark for servers. The result says overheads for write barriers of our implementation were less than 1%.

1. はじめに

近年，組み込み機器からサーバまで，産業界の幅広い分野でオブジェクト指向プログラミング言語 Java™☆ の利用が進んでいる．Java が普及した理由の 1 つとしてメモリ管理を自動化した点をあげることができる．C 言語など古典的なプログラミング言語ではソースプログラム中に `malloc()` や `free()` を明示的に記述してメモリの確保，解放を実施するが，手動によるメモリ管理では使用中のメモリ領域を誤って解放するといったバグがおこりやすい．これに対し，Java ではごみ集めという技法によってメモリ管理を自動化しており，メモリの確保解放にまつわるバグは発生しない．

一方，ごみ集めによるメモリ管理に固有な問題として停止時間がある．古典的な印掃式ごみ集めのアルゴリズム[11),18)] では，ごみ集めの開始にあたってプログラムの実行を進めるスレッド（mutator とよぶ）をすべて停止し，それらの実行時スタックや大域変数（これらを総括してルート領域とよぶ）中にあるポインタから再帰的にたどりうるデータ構造に印をつける．そして，印がつかなかったデータ構造を不要になったと見なして空き領域に還元し，mutaor の実行を再開する．つまり，ごみ集めを実行している間，mutator の実行は中断するが，中断する時間の長さはヒープ領域

† 日立製作所システム開発研究所
Systems Development Laboratory, Hitachi, Ltd.

☆ Java および HotSpot は米国およびその他の国における米国 Sun Microsystems, Inc. の商標です．

の大きさに比例する．昨今のサーバでは数ギガバイトのメモリを搭載しており，このような広いヒープ領域に古典的な一括実行型のごみ集めを適用すると，ごみ集めによる停止時間が数秒にわたるほど長くなり，アプリケーションの応答時間が劣化する．サーバ向けアプリケーションにとって応答時間は重要であり，ごみ集めによる間欠的な応答時間の劣化は好ましくない．したがって，サーバ向け Java 実行環境ではごみ集めに工夫をして停止時間を軽減する必要がある．

サーバに適したごみ集めについて考察する．まず，ごみ集めのアルゴリズムは，大まかに参照カウント式とトレース式の 2 種類に分類できる[8),18)]．参照カウント式ごみ集め[4)] には，ごみ集めによる長い停止時間が発生しない利点がある反面，参照の読み書きのたびにカウンタを操作するオーバヘッドが大きく，また，環状に参照しあうデータ構造群を空き領域にできない欠点がある．現在市場に流通している Java の実行環境は，サーバ向けかクライアント向けかを問わず，参照カウント式ではなく，実行効率に優れるトレース式ごみ集めを採用している．

トレース式ごみ集めはルート領域からポインタを再帰的にたどって到達しうるデータ構造を生きていると見なし，残りを不要になったものと見なして空き領域にする．つまりトレース式のごみ集めではデータ構造の生死を判定するためにデータ構造を再帰的にトレースするが，サーバアプリケーションなどではヒープ中に大量のデータ構造を配置するため，トレースに時間がかかって停止時間が問題になる．停止時間は次に示す 4 つの方法で短縮できる[13)]．

並列（parallel）ごみ集め トレース作業を複数のプロセッサで並列実行する[5)]．

並行（concurrent）ごみ集め トレース作業専用のスレッドを，mutator と同時に動かす[3),17)]．

漸次的（incremental）ごみ集め mutator が一定量のデータ構造を割り付けるたびに一定量のトレース作業を実施する[1),17)]．

ヒープ領域の分割 ヒープを複数の領域に分割し，1 回のごみ集めでは，そのうちの 1 つのみに限ってごみ集めを行うことで，トレースするデータ構造の量を制限し，停止時間の短縮を図る．

ヒープ領域を分割する方法の代表例として世代別ごみ集め[10),16)] や，世代別ごみ集めの旧世代領域をさらに分割する列車式ごみ集め[7)] をあげることができるが，本論文では世代別とは違った観点からヒープ領域を分割するスレッド別ごみ集め[6),9)] に焦点をあてる．本論文の 2 章ではスレッド別ごみ集めに関する過去の研究を示し，実行履歴を使うスレッド別ごみ集めについて述べる．続く 3 章では，実行履歴を使うスレッド別ごみ集めにおいて必要になるライトバリアの効率的な実現方法を提案し，4 章でライトバリアが実行速度に与える影響を評価する．5 章は結論である．

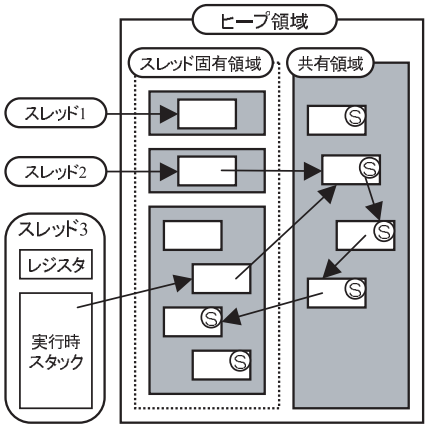

図 1 スレッド別ごみ集め
Fig. 1 Thread local garbage collection.

2. 関連研究

スレッド別ごみ集めでは，個々のスレッドが自身に固有なヒープ領域を，他のスレッドを止めずにごみ集めできる．スレッド別ごみ集めの利点を次に示す．

(1) 実行にあたって他のスレッドを止めない．このことは，アプリケーションサーバなど数十のスレッドが動作し，スレッドの停止自体に手間がかかる状況で有効に働く．

(2) スレッド固有のヒープに限ってごみ集めするので，ヒープ全体のごみ集めより停止時間が短い．

スレッド別ごみ集めの基本的なアイデアは，分散ごみ集め[8),15)] において計算ノードが単独で実施する局所的ごみ集めとよく似ている．本章ではまず，基本的なスレッド別ごみ集めの実現[6),9)] について述べ，次に実行履歴を使うスレッド別ごみ集め[6)] について述べる．

2.1 基本的なスレッド別ごみ集めの実現

スレッド別ごみ集めでは，次に示す 2 種類のヒープ領域を用意する（**図 1**）．

スレッド固有領域 個々のスレッドに切り分けて与えるための領域．この領域に配置するデータ構造は，基本的にデータ構造を割り付けたスレッドのみから参照でき，スレッド別ごみ集めの対象になる．

共有領域 基本的に複数のスレッドから参照できるデータ構造を配置するための領域．スレッド固有領域を含めた全域ごみ集めの対象となる．

割付け文がどちらの領域に割り付けるかは，割付け文ごとに定める．割付け文をスレッド固有領域に割り付けるものと，共有領域に割り付けるものに分類する方法としては次の 2 つがある．

Escape 解析を使う方法[9] Escape 解析[14] を使って，割付け文が割り付けるデータ構造が複数のスレッドから参照可能になりうるか調べ，なりうるならば☆共有領域に割り付ける．

実行履歴を使う方法[6] 割付け文が割付けたデータ構造が複数のスレッドから参照可能になる確率を，実行履歴を取得して求め，この確率が低くないならば共有領域に割り付ける．

実行時にメモリ管理系は，スレッド固有領域を切り分けて個々のスレッドに与える．スレッドは，割り付けの結果として自身に固有なヒープ領域が一杯になったら，該ヒープ領域を対象としてごみ集めを行う．このごみ集めをスレッド別ごみ集めとよぶ．スレッド別ごみ集めは，ごみ集めを行うスレッドが単独で実施可能であり，ごみ集めの実施にあたって他のスレッドを止める必要はない．なぜなら，スレッド別ごみ集めの対象となるのは，ごみ集めを行うスレッドのみが参照できるデータ構造だけだからである．

スレッドは，次の手順でスレッド別ごみ集めを行う．まず，スレッド固有のルート領域（スレッドの実行時スタックとレジスタ集合）から再帰的に参照をたどり，該スレッドのみが参照可能なデータ構造だけを経由して到達できる全データ構造に生存マークをつける．次に，該スレッドのみが参照可能なデータ構造のうち，生存マークがついていないデータ構造を死者と見なして解放する．

Escape 解析を使って割付け文を分類するスレッド別ごみ集めは，スレッド固有領域内の全データ構造をごみ集めの対象にできる．なぜなら Escape 解析を使う方法では，複数のスレッドから参照可能にならないと証明できたデータ構造のみをスレッド固有領域に配置するからである．

2.2 実行履歴を使うスレッド別ごみ集め

実行履歴を使って割付け文を分類するスレッド別ごみ集めでは，複数のスレッドから参照可能にならないと証明できないデータ構造もスレッド固有領域に配置する．この結果，Escape 解析を使う方法より多くのデータ構造をスレッド固有領域に配置できるが，スレッド固有領域内に複数のスレッドから参照可能なデータ構造が混じることがある．複数のスレッドから参照可能なデータ構造は，スレッド別ごみ集めの対象から除外する必要がある．なぜなら複数のスレッドから参照可能なデータ構造は，ごみ集めを行うスレッドに固有なルート領域から参照経由で到達不能であっても生きている場合があるからである．

```
0:  if (O_s != NULL && (is_shared(O_d)
1:     && (is_local(O_s))
2:   markRecursivelyAsShared(O_s);
3:  // 参照の書き込み
4:  O_d.field = O_s;
```

図 2 参照を書き込む前に実施する検査
Fig. 2 Implementation of write barrier.

実行履歴を使うスレッド別ごみ集めでは，複数のスレッドから参照可能なデータ構造をスレッド別ごみ集めの対象から除外するために，複数のスレッドから参照可能なデータ構造に印をつける．この印を本論文では共有マークとよぶことにする．図 1 では共有マークをⓈ印として表現した．共有マークは図 1 のようにデータ構造内に配置することもできるが，データ構造とは別の場所に配置することもできる．Domani らによるスレッド別ごみ集め実装では，共有マークをデータ構造とは別の場所に配置している[6]．共有マークのついたデータ構造は全域ごみ集めの際に生死を判定し，生きていたならば共有領域に移動する．データ構造に共有マークをつける作業は次のタイミングで行う．

(1) 共有領域にデータ構造を割り付ける割付け文が，割付けを行う際
(2) 共有マークのついていないデータ構造への参照を，共有マークのついているデータ構造に書き込む際

(2) のタイミングで共有マークをつけるために，実行履歴を使うスレッド別ごみ集めでは，参照の書き込みにライトバリアをかける．ライトバリアのついた参照を書き込む動作の実現を**図 2** に示す．図 2 の 0〜2 行目がライトバリアである．図 2 のコードは，0〜1 行目で書き込む参照 O_s が指示するデータ構造に共有マークがついておらず，かつ参照 O_d が指示する書き込み先のデータ構造に共有マークがついているか調べ，そうであれば 2 行目に進む．2 行目では関数 `markRecursivelyAsShared()` を呼び出し，O_s から参照を再帰的にたどって到達できる共有マークがつい

☆ Escape 解析は静的ごみ集め[2),12)] の対象となるデータ構造を生成する割付け文を探す場合にも利用する．静的ごみ集めとは，データ構造をスタックフレーム内に配置する技法である．Escape 解析の結果，静的ごみ集めの対象にはならないが，複数のスレッドから参照可能になることもないと分かったデータ構造がスレッド別ごみ集めの対象となる．

ていない全データ構造に共有マークをつける．

関数 `markRecursivelyAsShared()` は，割付け文をスレッド固有領域に割り付けるものと，共有領域に割り付けるものに分類する役割も担う．分類はデータ構造に共有マークをつける際に，次の手順で実施する．

(1) データ構造を割り付けた割付け文を求める．
  - データ構造から，該データ構造を割り付けた割付け文を算出可能にするために，データ構造を割り付ける際に，データ構造のヘッダに割付け文固有の番号（割付け文 ID）を記載しておくものとする．

(2) 求めた割付け文が複数のスレッドから参照可能なデータ構造を割り付けた回数に 1 を加算する．

(3) 加算の結果，スレッド別ごみ集めが定める閾値を超えたならば，割付け文を共有領域への割付け文に書き換える．この閾値を本論文では共有化閾値とよぶことにする．
  - 割付け文は初期状態においてはスレッド固有領域に割り付けるものとする．ただし，最初から複数のスレッドが参照できるデータ構造を割り付ける割付け文については，初期状態から共有領域への割り付けるものとする．たとえば Java では `java.lang.Class` や `java.lang.Thread` クラスのインスタンスは最初から複数のスレッドが参照できる．

3. ライトバリアの高速化

本論文では，実行履歴を使うスレッド別ごみ集めにともなうオーバヘッドを軽減する方法を提案する．スレッド別ごみ集めにともなうオーバヘッドには，ヒープの初期化時などに発生する一時的なものと，プログラムの実行中に発生しつづける定常的なものがある．定常的なオーバヘッドの発生源は次の 2 つである．

(1) ライトバリア（図 2 の 0〜2 行目）

(2) インスタンスの割付け時における共有マークの初期化と割付け文 ID の設定

ライトバリアについては，4 章で述べるように，2 行目の関数 `markRecursivelyAsShared()` を呼び出すことは稀である．したがってライトバリアについては 0〜1 行目が定常的なオーバヘッドの発生源となる．

3.1 共有マークおよび割付け文 ID のヘッダへの配置

本論文では定常的オーバヘッドを軽減するために，共有マークおよび割付け文 ID をデータ構造のヘッダに配置する方法を提案する．ただヘッダはごみ集め以外にも様々な用途に利用する領域であるので，ヘッダにこれらの値を格納するにあたっては，ヘッダを利用するごみ集め以外の機能にも改変が必要になりうる．

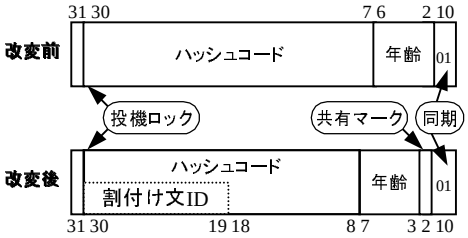

図 3 ヘッダのフォーマット
Fig. 3 Header format.

ヘッダの用途はごみ集めを提供する言語処理系によって異なるので，ヘッダに共有マークおよび割付け文 ID を配置するにあたって発生する障害について，一般的に議論することは難しい．そこで本論文では Sun Microsystems 製の Java Virtual Machine（VM）である HotSpot VM をとりあげ，HotSpot VM 向けに実行履歴を使うスレッド別ごみ集めを実装する場合の問題点とその解決策を示す．

HotSpot VM がヒープ上に配置するデータ構造につけるヘッダのフォーマットを**図 3** 上段に示す．図 3 下段は実行履歴を使うごみ集め向けに改変したヘッダのフォーマットであり，割付け文 ID と共有マークを記録するための領域が増えている．割付け文 ID はハッシュコードの上位桁を兼ねる．ヘッダに割付け文 ID と共有マークを配置することの利点と問題点について述べる．まず，利点を次に示す．

- インスタンス割付け時における共有マークの初期化と割付け文 ID の設定にかかるコストが 0〜1 命令で済む．
  - HotSpot VM は元々，インスタンス割付け時にヘッダに初期値として 1 を書き込む．ヘッダを改変したあとではヘッダの初期値に割付け文 ID と共有マークの初期値が加わる．共有マークの初期値は，スレッド固有領域への割付け文では 0，共有領域への割付け文では 1 になる．
    改変後のヘッダの初期値はコンパイル時に計算できるから，ヘッダの改変前後におけるインスタンス割付けのコンパイル済みコードの変化は，書き込む値が定数 1 から上記の定数に変わるだけである．この変化にともなう命令数の増加は，プロセッサによって異なるが，サーバや PC 向けのプロセッサでは 0〜1 命令である．
- 図 2 のライトバリアで使うマクロ `is_local()` や

is_shared() は単にヘッダに書いてある値を参照して下から 3 bit 目を検査するだけの命令列で実現できる．

– Domani らのスレッド別ごみ集めでは共有マークをデータ構造とは別の場所に配置しており[6]，その分，マクロ is_local() や is_shared() の実現に必要な命令数が多い．

一方でヘッダに割付け文 ID と共有マークを配置することには次の問題点がある．

- HotSpot VM ではモニタを確保する際にヘッダを書き換える．このことは，図 2 のライトバリアで使うマクロ is_local() や is_shared() の実装するうえで問題になる．これらのマクロはヘッダ上にある共有マークを参照するので，書き換えによってヘッダから共有マークが消失すると正常に動作しなくなるからである．
- 割付け文 ID を記録するための領域を広く確保できない．この結果，大規模なアプリケーションプログラムでは割付け文 ID が不足し，実行履歴を利用するごみ集めを利用できなくなりうる．

これらの問題の解決策について順次述べる．

3.2 モニタの実装の改変

モニタの確保にともなうヘッダの書き換えに対する対策について述べる前に，まず，スレッド別ごみ集め向けの改変を加えていないオリジナルの HotSpot VM におけるモニタの確保手順を次に示す．

- モニタを確保前の状態では，**図 4** 上段に示すように，データ構造のヘッダに図 3 上段に示すフォーマットのデータが記録してある．
- モニタを確保する際には，このデータを実行時スタック上のモニタ構造体に退避し，ヘッダの内容をモニタ構造体のアドレスで上書きする（図 4 下段）．
- モニタを解放する際には，モニタ構造体に退避したデータをヘッダに書き戻す．

モニタ確保前にヘッダに記録してある値の下 2bit は 01 になっているが，これはモニタを確保している状態と，していない状態を区別する際に利用する．モニタ構造体は 4 バイト境界に配置してあるので，そのアドレスの下 2 bit は 00 である．したがってデータ構造のヘッダを参照して，下 2 bit が 01 であればモニタを確保していない，00 であればモニタを確保している，より正確にはヘッダに書いてあるアドレスを実行時スタックに含むスレッドがモニタを確保していると判断できる．

ライトバリアの実現にあたっては，モニタの実装が問題になる．なぜならモニタを確保する際にヘッダの内容をスタック上のモニタ構造体内に退避するが，このとき共有マークも一緒に退避してしまうからである．ライトバリアにおいて，共有マークがデータ構造の先頭についているか，それともスタック上にあるか判別することは不可能ではないが，判別の過程で atomic 演算が必要になり，大きなオーバヘッドがかかる．

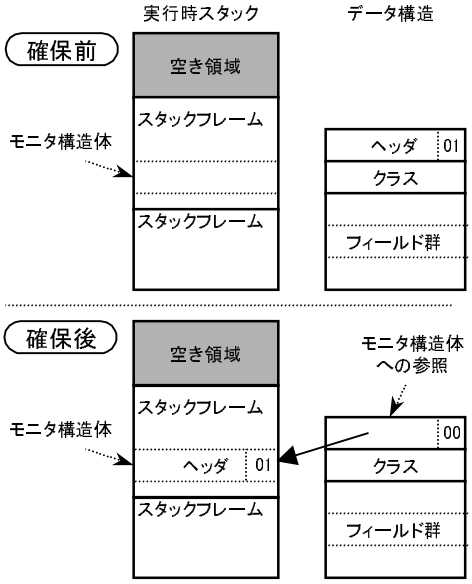

図 4 モニタの確保
Fig. 4 Locking an object.

そこで本論文ではモニタの確保にあたり，ヘッダに書き込むモニタ構造体のアドレスに共有マークを移植する手法を提案する．この方法では，モニタの確保および解放の際に 1，2 個の論理演算命令を実行するだけで，共有マークをつねにヘッダ上に配置可能になり，したがってマクロ is_local() や is_shared() を単にヘッダに書いてある値を参照して下から 3 bit 目を検査するだけの命令列で実現できる．提案する技法の実施手順を次に示す．

(1) スタック上のモニタ構造体は 8 バイト境界に整列して配置する．この結果，モニタ構造体のアドレスの下 3 bit は必ず 000 になる．
(2) 共有マークはヘッダの下から 3 bit 目に配置する．
(3) モニタの確保にあたり，ヘッダにモニタ構造体のアドレスを上書きする際には，モニタ構造体のアドレスそのものを書き込むのではなく，モニタ構造体のアドレスと，上書き前のヘッダについている共有マークの論理和を書き込む．
(4) モニタの解放にあたり，ヘッダに書き込んでおいたモニタ構造体のアドレスが必要になったら，ヘッダに書き込んでおいた値の下から 3 bit 目

表 1 実行履歴を使うスレッド別ヒープの性能
Table 1 Performance of thread local garbage collection using runtime profile.

ベンチマーク名称	総割付量（MByte）	実局所率（%）	総局所率（%）	検出率（%）	速度劣化率（%）	速度劣化要因の実行頻度（回/ms） ライトバリア	ヘッダ初期化	共有マーク移植
SPECjbb2000	1,158	30	57	57	−0.19	546	150	485
VolanoMark								
サーバ側	352	86	93	93	0.47	11	27	81
クライアント側	514	42	95	44		31	28	80
SPECjvm98								
_201_compress	105	100	100	100	−0.01	0	1	59
_202_jess	293	8	100	8	0.50	736	740	457
_209_db	75	88	100	89	2.10	667	71	1,250
_213_javac	205	29	77	38	0.06	812	247	633
_222_mpegaudio	1	46	70	65	−0.04	198	1	97
_227_mtrt	159	98	100	98	0.99	374	827	88
_228_jack	176	31	96	33	0.11	685	382	957

をクリアすることで，モニタ構造体のアドレスを復元してから使う．

3.3 割付け文 ID のの領域長

図 3 下段では割付け文 ID のために 12bit の領域を確保している．12 bit の割付け文 ID で判別できるのは，たかだか $2^{12} = 4{,}096$ 個の割付け文にすぎないから，大規模なアプリケーションでは割付け文 ID が不足しうる．そこで我々は関数 `markRecursivelyAsShared()` においてデータ構造を生成した割付け文を検索する際に，割付け文 ID のほかにクラスも検索鍵として利用することにした．この結果，4,096 という数はクラスあたりの区別できる割付け文の最大数になり，判別できる割付け文の数が大幅に増えた．

ヘッダにおける割付け文 ID の領域長を増やせば区別できる割付け文の数も増えるが，割付け文 ID の領域はハッシュコードの一部でもあるので，割付け文 ID の領域長を増やすとハッシュコードの性質が悪くなる．すなわち，図 3 下段のヘッダにおいて，ハッシュコードの上位 12 bit は割付け文 ID と兼用であるため，同じ割付け文が生成したデータ構造であれば，この部分はつねに同じ値になる．下位 11bit に乱数を入れているので，同じ割付け文が生成したデータ構造だからといってハッシュコードが完全に同一になることはないが，割付け文 ID の領域を増やせば乱数の領域が減ってしまう．割付け文 ID の領域長を 12 bit としたことの妥当性については次章で考察する．

4. 評　　価

本章ではまず，実行履歴を使うスレッド別ごみ集めにおいて定常的に発生するオーバヘッドが実行速度に与える影響を評価し，次に割付け文 ID の領域長について考察する．

評価対象としては，サーバ向け Java アプリケーションのベンチマークである SPECjbb2000 および VolanoMark 2.5.0.9 とした．参考までに SPECjvm98 も評価したが，SPECjvm98 のように起動するスレッド数の少ないクライアント向けアプリケーションはスレッド別ごみ集めの恩恵を受けない．評価に利用した機械は Hitachi 9000 V（PA-8500 400 MHz，メモリ 512 MByte）である．また，Java VM は HotSpot VM を我々が HP-UX®☆向けに移植したものを利用した．

4.1 定常的オーバヘッドが実行速度に与える影響

実行履歴を使うスレッド別ごみ集めでは，実行時に次のオーバヘッドが定常的に発生する．

(1) ライトバリア（図 2 の 0〜2 行目）
(2) インスタンスの割付け時におけるヘッダの初期化（割付け文 ID と共有マーク）
(3) モニタの確保時などにおける共有マークの移植

これらの処理が実行速度に及ぼす影響を評価した結果を**表 1** の速度劣化率の欄に示す．速度劣化率が正の値であることは，実行速度が劣化したことを意味する．評価は次の 2 つの HotSpot VM でベンチマークを実行することにより行った．

- オリジナルの VM
- 共有マークを管理する機能を追加した VM

後者の VM は共有マークを管理するための一連の処理を行い，たとえば後者の VM の動的コンパイラはライトバリアなど，(1)〜(3) の処理を実施するためのコードを生成する．ただし後者の VM は共有マークの管理をするだけで，ごみ集め自体はオリジナルの HotSpot VM と同じ全スレッド停止型のものを使う．

☆ HP-UX は米国 Hewlett Packard, Inc. の登録商標です．

したがって評価に使う 2 つの VM の定常的な違いは (1)〜(3) の処理に関するオーバヘッドを被るか否かのみである．

なお，後者の VM のコンパイラには，ライトバリアに関する若干の最適化を実装してあり，簡単なフロー解析によって書き込む参照が必ず null になると分かる参照書き込みの前にはライトバリアを挿入しない．

表 1 から，速度劣化率の相乗平均を求めると，0.44% であることが分かる．0.44%という値は測定誤差の範囲内であり，個々の測定結果が純粋に速度劣化率を表すとは考えにくい．表 1 で速度劣化要因の実施回数が多いベンチマーク項目が必ずしも速度劣化率が高いとはいえず，たとえば _209_db と _228_jack では速度劣化要因の実行頻度が比較的よく似ているが☆，速度劣化率は _209_db の 2.1%に対して _228_jack では 0.11%と大きな違いがある．しかしながら総覧すれば速度劣化率はおおむね 0〜1%の範囲にあり，相乗平均としての 0.44% という数値はおおむねの目安になると考える．

なお，表 1 の速度劣化率を計測するにあたっては，共有化閾値を 0% とした．つまり，1 度でも複数のスレッドから参照可能なデータ構造を割り付けた割付け文は，共有領域に割付けを行うようにした．共有化閾値を 0%より大きくすると，速度劣化率が表 1 に示した値より大きくなる可能性がある．なぜなら共有化閾値を大きくすると，複数のスレッドから参照できるデータ構造をスレッド固有領域に配置し，その結果としてライトバリアにおいて関数 `markRecursivelyAsShared()` を呼び出す機会が増えるからである．

共有化閾値が 0%のスレッド別ごみ集めにおいて，関数 `markRecursivelyAsShared()` がベンチマークの実行期間中，いつどれだけのデータ構造に共有マークをつけたか調査した結果を**図 5** に示す．図 5 の横軸は実行時間全体を 100%とした場合に，実行開始から何%の時間が経過したかを表し，縦軸は 5%ごとに区切った横軸の各区間において `markRecursivelyAsShared()` が何個のデータ構造に共有マークをつけたかを示す．図 5 から共有マークをつける処理はアプリケーション起動時に大量に発生するが，他はあまりおきないことが分かる☆☆．したがって共有化閾値が 0%のスレッド別ごみ集めは，

☆ ライトバリアの実施回数がほぼ同一．ヘッダの初期化と共有マークの移植頻度には違いがあるが，前者を実行するのが算術演算命令 1 つで後者が論理演算命令 2 つであることを考慮すると，2 者を合計した負荷は似たような値になると推測できる．

☆☆ 冗長さを避けるため，図 5 には SPECjbb2000 と VolanoMark（サーバ側）のみ掲載したが，他のベンチマークもおおむね同様の傾向を示す．

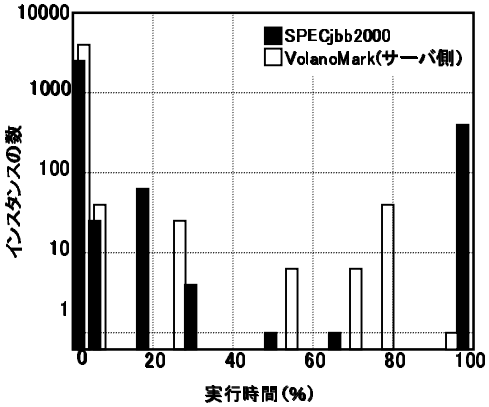

図 5 `markRecursivelyAsShared()` で共有マークをつけたインスタンスの数
Fig. 5 Number of instances `markRecursivelyAsShared()` marked.

定常状態において関数 `markRecursivelyAsShared()` のオーバヘッドをほとんど被らないといえる．

なお，共有化閾値は実行履歴を使うスレッド別ごみ集めが処理できるデータ構造の量にも影響を与える．共有化閾値を大きくすると，スレッド別ごみ集めで処理できるデータ構造の量が増え，その反面，`markRecursivelyAsShared()` の実行回数が増えて実行速度が劣化する．参考までに，共有化閾値が 0%の実行履歴を使うスレッド別ごみ集めが処理できるデータ構造の割合を評価した結果を表 1 の実局所率の欄に示す．表 1 における実局所率，総局所率，検出率という用語の意味を次に示す．

実局所率 スレッド別ごみ集めの実装（ここでは共有化閾値が 0%の実行履歴を使うスレッド別ごみ集め）がスレッド固有のヒープ領域に配置するデータ構造の量（バイト数）が総割付量に占める比率．

総局所率 クラス変数への参照の書き込みなどにより複数のスレッドから参照できる状態に 1 度もならずに死亡したデータ構造（スレッド局所データ構造）が総割付量に占める比率．つまり，理論上はスレッド固有のヒープ領域に配置できるデータ構造の比率（ただし，他スレッドから参照可能になったデータ構造は，実際には他スレッドから参照を受けずに死亡した場合も，あとから単一スレッドのみが参照可能な状態に戻った場合も，スレッド局所データ構造として扱わない）．共有化閾値が 100%である場合，つまりすべての割付け文がつねにスレッド固有領域に割付けを行う場合における実局所率に一致する．総局所率とスレッドの多重度が高いアプリケーションにはスレッド別ごみ

集めを適用する意味がある．

検出率 実局所率を総局所率で除した値．検出率が高いほど優れたスレッド別ごみ集めの実装である．

表 1 から，SPECjbb2000 で 30%，VolanoMark（サーバ側）で 86%の実局所率を確保できることが分かる．共有領域の消費速度はスレッド固有のヒープ領域に配置できないデータ構造の量に比例し，共有領域のごみ集めを実行する間隔は共有領域の消費速度に反比例することから，共有化閾値が 0%の実行履歴を使うスレッド別ごみ集めでは，共有領域のごみ集めを実行する間隔を SPECjbb2000 では $\frac{1}{1-0.30}-1=43\%$，VolanoMark では 614%長くできると推測できる．

最後になるが，速度劣化率はコンパイラに最適化を追加すれば表 1 に示した値より小さくなる．たとえば図 2 の検査について，フロー解析によって書き込む参照が絶対に null でないと分かった場合に，書き込む参照が null か検査する分岐を除去する最適化などは未実装であり，今後評価したい．

4.2 割付け文 ID 領域長に関する考察

ベンチマークを通じて同一クラスのインスタンスの割付け文の数が最も多いものを求めると，`_222_mpegaudio` には short 型配列の割付け文が 701 カ所ある．小規模ベンチマークで 701 カ所もあるのでは，上限が $2^{12}=4{,}096$ では不足するとも考えられる．しかし，これら 701 個の割付け文について調査してみると，個々の実行回数は 1 回である場合がほとんどであり，割付け先をスレッド固有領域にすべきか検討に値する割付け文を 1 つも含まないことが分かった．このことから，割付け文 ID を節約するうえで，割付け回数が多い割付け文に限って ID を符番する技法が有効であろうと推測できる．

最も簡単な ID を与える基準は「n 回以上割り付けたら ID を与える（ID を受け取る前は無効な ID をつけたデータ構造を共有領域に割り付ける）」というものだろう．**表 2** に示すように $n=2$ とするだけでも，ID を与える割付け文の数はすべてに与える場合と比較して，必要になる ID の個数が 12〜57%，相乗平均で 29%になることが分かる．

4.3 Escape 解析によるスレッド別ごみ集めの有効性

最後に，本論文で議論した実行履歴を使うスレッド別ごみ集めとは異なる，Escape 解析を使うスレッド別ごみ集めの有効性について考察する．Escape 解析は実行にかかるコストが小さくないので実行時コンパイラで実施する場合には解析対象を絞る方が好ましいが，解析対象を絞るとスレッド固有ヒープにデータ構造を割り付ける文の数も減る．解析対象をどれだけ絞ることができるか考察するために，どれだけの割付け文を解析したら最大何%のデータ構造をスレッド固有ヒープに配置できるか評価した結果を**図 6** に示す．

表 2 n 回以上実行した割付け文
Table 2 Allocation statements executed no less than n times.

ベンチマーク名称	総数	2 回以上（%）	10 回以上（%）
SPECjbb2000	3,833	34	20
VolanoMark			
サーバ側	3,773	32	15
クライアント側	913	27	19
SPECjvm98			
_201_compress	837	22	9
_202_jess	1,027	28	15
_209_db	843	21	7
_213_javac	1,662	57	45
_222_mpegaudio	1,823	12	3
_227_mtrt	945	31	16
_228_jack	1,083	33	24

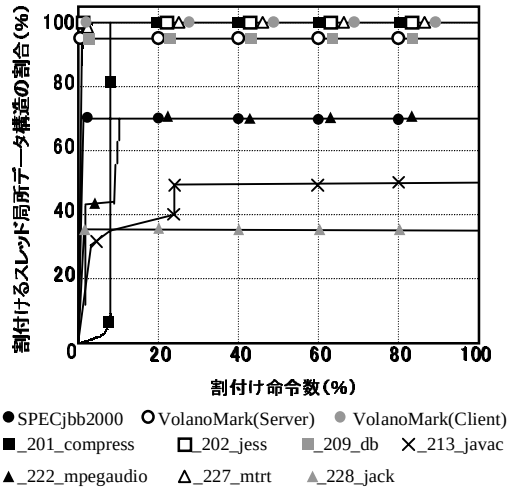

図 6 スレッド局所データ構造の割付け文に対する分布
Fig. 6 Distribution of thread local data structure against allocation statement.

図 6 のグラフは，割付け文を実行回数の多いものから順に並べ，それらのうちスレッド局所データ構造のみ生成したものの割付け量の積算値（縦軸，厳密には縦軸は積算値が総割付量に占める比率）と割付け文の数（横軸）の関係をプロットしたものである．すなわち実行時にスレッド局所データ構造のみ生成した割付け文のすべてについて Escape 解析を適用すれば，スレッド局所データ構造のみを生成すると証明できると仮定した場合の積算値と割付け文の関係であり，実際にはすべて証明できるわけではないから実際の割付け

量は図 6 に示した積算値より小さくなりうる☆.

図 6 のグラフは，立ち上がりが急峻であるほど，より少ない解析量でより多くのデータ構造をスレッド固有ヒープに配置できることを意味する．図 6 から，ほとんどのベンチマーク項目において数%の割付け文を解析すれば，ほぼ上限値のデータ構造をスレッド固有ヒープに配置できることが分かる．したがって Escape 解析による分類法が有効になるか否かの鍵は，どれだけの割付け文についてスレッド局所データ構造のみ割り付けると証明できるかにかかっているといえる．

5. 結　　論

本論文では実行履歴を使うスレッド別ごみ集めにおけるライトバリアのオーバヘッドを軽減する方法として，割付け文 ID と共有マークをヘッダに配置する方法を提案した．サーバ向けベンチマーク SPECjbb2000 を使って評価した結果，本論文で提案した方法により，ライトバリアなどから発生する実行速度の定常的な劣化を 1%未満にできることが分かった．

参 考 文 献

1) Baker, H.G.: List processing in real-time on a serial computer, *Comm. ACM*, Vol.21, No.4, pp.280–294 (1978).
2) Barth, J.M.: Shifting garbage collection overhead to compile time, *Comm. ACM*, Vol.20, No.7, pp.513–518 (1977).
3) Bohem, H.-J., Demers, A.J. and Shenker, S.: Mostly parallel garbage collection, *ACM SIGPLAN Notices*, Vol.26, No.6, pp.157–164 (1991).
4) Coollins, G.E.: A method for overrapping and erasure of lists, *Comm. ACM*, Vol.3, No.12, pp.655–657 (1960).
5) Dimpsey, R., Arora, R. and Kuiper, K.: Java server performance: A case study of building efficient, scalable JVMs, *IBM Systems Journal*, Vol.39, No.1, pp.151–174 (2000).
6) Domani, T., Goldshtein, G., Kolodner, E.K., Lewis, E., Petrank, E. and Shenwald, D.: Thread-Local Heaps for Java, *International Symposium on Memory Management 2002*, pp.76–87 (2002).
7) Hudson, R.L. and Moss, J.E.B.: Incremental Collection of Mature Objects, *International Workshop on Memory Management 1992*, pp.388–403 (1992).
8) Jones, R. and Lins, R.: *Garbage Collection Algorithms for Automatic Dynamic Memory Management*, John Wiley and Sons, West Sussex, England (2000).
9) King, A.C.: Removing GC Synchronization, *OOPSLA 02 COMPANION*, pp.112–113 (2002).
10) Lieberman, H. and Hewitt, C.E.: A real-time garbage collector based on the lifetime of objects, *Comm. ACM*, Vol.26, No.6, pp.419–429 (1983).
11) McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, *Comm. ACM*, Vol.3, pp.184–195 (1960).
12) McDowell, C.E.: Reducing garbage in Java, *ACM SIGPLAN Notices*, Vol.33, No.9, pp.84–86 (1998).
13) Ossia, Y., Ben-Yizhak, O., Goft, I., Koolodner, E.K., Leikehman, V. and Owshanko, A.: A Parallel, Incremental and Concurrent GC for Servers, *PLDI 2002*, pp.129–140 (2002).
14) Park, Y.G. and Goldberg, B.: Escape analysis on lists, *ACM SIGPLAN Notices*, Vol.27, No.7, pp.116–127 (1992).
15) 市吉伸行：分散ごみ集め，情報処理，Vol.35, No.11, pp.1027–1032 (1994).
16) 中西正和，田中詠子：世代別ごみ集め，情報処理，Vol.35, No.11, pp.1014–1019 (1994).
17) 湯淺太一：実時間ごみ集め，情報処理，Vol.35, No.11, pp.1006–1013 (1994).
18) 日比野靖：ごみ集めの基本アルゴリズム，情報処理，Vol.35, No.11, pp.992–999 (1994).

(平成 15 年 9 月 22 日受付)
(平成 15 年 12 月 2 日採録)

千葉　雄司（正会員）

1972 年生．1997 年慶應義塾大学大学院理工学研究科計算機科学専攻修士課程修了．同年日立製作所（株）入社．システム開発研究所にてコンパイラの研究開発に従事．2001 年より 2003 年まで慶應義塾大学非常勤講師を兼務．2003 年から中央大学客員研究員を兼務．

☆ ただし図 6 ではインライン展開によって割付け文を多重化し，スレッド局所データ構造のみを生成すると証明できる割付け文の数を増やす技法の影響までは考慮していないので，積算値が図 6 より大きくなる可能性もある．