Vol. 44 No. SIG 13(PRO 18) 情報処理学会論文誌：プログラミング Oct. 2003

動的に割付け戦略を最適化するJavaメモリ管理機構

中村 実† 前田 宗則† 小沢 年弘†

エンタープライズ分野のJavaプログラムは，マルチスレッド処理を行っている．このようなプログラムは，生成スレッド以外から参照されないスレッドローカルなオブジェクトを大量に生成する傾向がある．そのためJavaVMがスレッドローカルなオブジェクトをあらかじめ区別して，スレッド固有のメモリ領域に割り当てることで，スレッドごとの非同期なガーベージコレクション（GC）を行うことができ，実行性能を改善できると考えられる．既存のスレッドローカルGCの手法は，スレッドローカル性の判定を静的コード解析を用いていた．しかし静的コード解析には多くの時間が必要で，十分なオブジェクトを抽出することも難しい．そこで我々は，実行時情報を用いてオブジェクトの割り付けを最適化するスレッドローカルGCの手法を提案する．本手法の特徴は，オブジェクトのスレッドローカル性の判定が困難な場合でも,「投機的」にスレッド固有メモリ領域への割当てが可能な点にある．他のスレッドから到達可能となったオブジェクトは「投機失敗」と見なされ，スレッドローカルGCによる回収を行わない．スレッドローカル性の判定は，他のオブジェクトからの参照の有無によって行い，参照を再帰的に手繰る必要がない．それゆえ判定処理は非常に軽量である．また，ランタイムは投機成功・失敗の履歴情報を用いて，アロケート命令の割付け戦略を動的に決定していく．本論文では提案手法の実装と評価を行い，8〜16 CPUのSMPマシンにおいてスケーラビリティの向上を得た．

Speculative Allocation: Thread-independent Memory Management for Java

Minoru Nakamura,† Munenori Maeda† and Toshihiro Ozawa†

Thread-local memory management is suited to multi-threaded programs which create bulk, short-lived and thread-private objects, such as mission-critical and enterprise-scale Java applications, since it may eliminate the synchronizations for object allocations and garbage collection(GC) among threads. Existing thread-local GC schemes are based on static code analysis to decide “purely” thread-local objects all through their lives. This approach has no runtime overhead but exhaustive, and may only decide insufficient amount of objects by its strictness. In this paper, we propose a new thread-local GC scheme which supports “speculative allocation” to the thread-specific memory for arbitrary objects, instead of heavy analyses. Objects supposed to be thread-local upon their allocation are allocated in the thread-specific memory, and then keep their thread-local state unless their references are stored into other objects’ fields. The thread-locality condition is a non-recursive procedure and ends in fixed steps per store operation. Moreover, to reduce store operation and GC overhead, allocation operations having created non-local objects are dynamically re-written to alternative shared memory allocation operations. Therefore, the locality checking has quite low overhead. We present implementations and evaluations of this method. The results show that it provides some performance improvement for large scale SMP machine such as from 8 to 16 ways.

1. はじめに

近年，エンタープライズ分野でJavaが広く使われるようになってきている．これはJavaがガーベージ・コレクション（GC）と呼ばれる自動的なメモリ管理システムを導入したことにより，メモリリークや誤ったメモリ解放によるバグの発生を防止できる点が大きい．しかし，GCはプログラマによる直接的なメモリ管理方法に比べ，プログラムごとに柔軟な制御を行うことは困難であり，メモリ管理のオーバヘッドで性能低下が生じることがしばしばある．

エンタープライズ分野のアプリケーションは，複数のスレッドがトランザクション単位で処理を進めるものが多い．トランザクション内で生成されるほとんどのオブジェクトは，トランザクション内で閉じた寿命を持ち，他のスレッドからアクセスされないものが多数存在する．このようなアプリケーションは，スレッ

† 株式会社富士通研究所
FUJITSU LABORATORIES LTD.

ド別のメモリ管理を適用することで実行性能を改善することが可能である．

スレッド別メモリ管理方式はこれまでにいくつか提案，実用化がなされている[1)~3)]．その1つにスレッドローカルGC[2)~3)]の手法がある．オブジェクトの中には生成スレッド以外のスレッドからつねに到達不能なものが存在する．これをスレッドローカルオブジェクトと呼ぶ．スレッドローカルGCは，スレッド固有メモリ領域を設け，そこにスレッドローカルオブジェクトのみを割り付けることで，各スレッドが非同期にGCを行うことを可能にするものである．そのため，既存の研究では，エスケープ解析（escape analysis）[4)~6)]などの静的コード解析を用いて，スレッドローカルなオブジェクトを検出していた．しかし，解析的手法ではコストの大きい関数間解析が必要になる．また，スレッドローカルと判定できるオブジェクトも十分には抽出できない．

そこで本論文では，スレッドローカルオブジェクトの抽出をランタイム支援によって行うことで，スレッドローカルGCを実現する機構を提案する．解析的手法も併用するが，基本ブロック内に限定された軽量な解析のみに抑え，静的コード解析のコストを大幅に削減する．本機構をスレッドローカルヒープ（Thread-local Heap; TLH）と呼ぶ．

TLHがターゲットとするのは，SMP型マルチプロセッサマシン上で動作するマルチスレッドプログラムである．スレッドローカルGCを用いることによって，スレッド数分の並列度を得ることができ，実行性能の改善が期待できる．

我々はこのTLHをプロダクトレベルのJava仮想マシンであるSun Hotspot VM 1.4.0上に実装し，性能評価を行った．

本論文では，2章でTLHのアルゴリズムを述べる．3章で実装のベースとしたSun Hotspot VMと実装方法について述べたあと，4章で評価を行う．関連研究との比較を5章で行い，最後に6章でまとめを述べる．

2. アルゴリズム

ランタイム支援によって，オブジェクトのスレッドローカル性を動的に判断するには，以下のような手法が必要となる．まず，オブジェクトがスレッドローカルであると仮定して，スレッド固有メモリ領域に割付けを行う（投機的割付け）．投機的に割り付けたオブジェクトのうち，他のスレッドから到達可能になったものを検出し，投機失敗と判定する．スレッドローカルGCは投機失敗オブジェクトを誤って回収しないように保護し，真のスレッドローカルオブジェクトのみを回収する．

以下，2.1節で投機的割付けを用いたスレッドローカルGCの方式について検討する．2.2節では，この方式が想定するJavaランタイムのメモリ構成について述べる．2.3節では投機失敗の検出方法であるライトバリア処理の詳細を述べる．2.4節，2.5節では投機失敗オブジェクトを保護しながらスレッドローカルGCを行う手法を述べる．2.6節では投機失敗を削減する最適化手法について述べる．

2.1 方式の検討

実行時にスレッドローカル性のチェックを行う手法が3通り考えられる．1つ目は参照読み込み時にチェックを行い，読み込んだ参照が自スレッドで生成されたものかどうかをチェックする方法．2つ目は参照書き込み時にチェックを行い，書き込んだ参照が自スレッドで生成されたものかどうかをチェックする方法．3つ目は分散GCなどに用いられる輸出表を用いる方法である．

一般のプログラムでは，参照読み込みは書き込みよりも頻度が多いため，参照読み込み時のスレッドローカル性のチェックは書き込み時のチェックよりコストが大きい．また，輸出表を用いる方法は参照の読み込みと書き込みの両方にチェックが必要なため，さらにコストが大きくなる．

そこで，参照書き込み時にチェックを行う戦略を採用する．このとき，以下のような機構が必要となる．

(1) 投機的に割り付けたオブジェクトの投機失敗を検出する機構
(2) スレッドローカルGCからの投機失敗オブジェクトを保護する機構
(3) 投機失敗が予測されるオブジェクトをスレッド共有ヒープ領域へ割り付けることによる投機失敗を軽減する機構

最も問題となるのは(1)である．スレッドローカルオブジェクトとそれ以外のオブジェクト（グローバルオブジェクト）を完全に分離するような検出は，コストが非常に大きいものになる．なぜなら投機的割付けオブジェクトが他の投機的割付けオブジェクトを参照している場合，参照元の投機的割付けオブジェクトが投機失敗とになった場合，その参照先のオブジェクトも再帰的に投機失敗となるからである．

そこで，我々の提案するTLHでは，スレッドローカルオブジェクトのすべてをスレッドローカルGCの対象とはせずに，投機の失敗の検出が容易な部分集合

のみを扱う．TLH のスレッドローカル GC で回収の対象とするのは，自分自身または他のオブジェクトによって参照されないスレッドローカルオブジェクトのみとする．

Java は，オブジェクトから参照されないスレッドローカルオブジェクトは，それを生成したスレッドのスタックからのみ直接参照される．そこで，このようなスレッドローカルオブジェクトをスタック直接参照オブジェクトと呼ぶことにする．また Java のオブジェクトはすべてヒープ空間に置かれるので，オブジェクトから参照されているオブジェクトは，ヒープ内にその参照が書き込まれている．そこで，このようなオブジェクトを被ヒープ参照オブジェクトと呼ぶことにする．

オブジェクトの参照がヒープに書き込まれるかどうかの判定は，Java の参照書き込み命令をチェックすることで簡単に行うことができる．また，スタック直接参照オブジェクトがガーベージであるかどうかは，オーナースレッドのスタックを走査し，その中にオブジェクトへの参照が含まれているかどうかをチェックするだけで決定できる．これは，オブジェクトが生存しているかどうかの完全な判定が，ルートから参照を再帰的にたぐる処理を必要とするのに比べて，非常に軽量である．

投機失敗オブジェクトは，最終的にスレッド固有メモリ領域からスレッド共有メモリ領域に移動させる必要がある．ただし，他のスレッドから参照されているオブジェクトの移動は，一般に他のスレッドをすべて停止させてから行う必要がある．そこで，TLH ではこの投機失敗オブジェクトの移動を限界まで遅延させ，スレッドローカル GC では投機失敗オブジェクトをスレッド固有メモリ領域内に放置する．スレッドローカル GC 後は，投機失敗オブジェクトの占めている領域を避けながら割り付けていく．

スレッド固有メモリ領域が投機失敗オブジェクトによって飽和した場合には，これを回復させる特別な GC を発生させる．この GC では，すべてのスレッドを停止させて投機失敗オブジェクトをスレッド共有メモリ領域へ移動させることになる．

この方法は，静的・動的な最適化によって投機失敗オブジェクトの発生を十分に抑えることが可能であれば，メモリ効率と実行効率の向上を得られると考えられる．

2.2　メモリ構成

TLH は，以下のようなシステム構成を想定する（**図 1**）．ユーザプログラムは複数のスレッドによって実行され，システム内に存在するメモリ空間は，Java スレッドごとに存在するスレッド固有メモリ領域と，複数の Java スレッドによって共有される共有ヒープに分けられる．共有ヒープは，本アルゴリズムによって直接管理されない領域である．本論文ではスレッド固有メモリ領域を Local Object Strage（LOS）と呼ぶ．LOS は投機失敗オブジェクトの占める領域を避けて再利用する必要がある．そのためフリーリスト管理を行う．

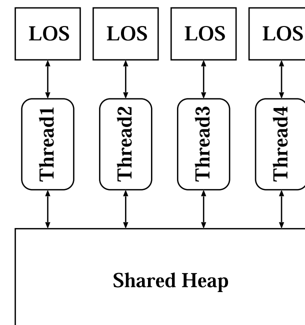

図 1　メモリ構成
Fig. 1　Memory architecture.

LOS 割付けを行う場合，フリーリストから要求サイズに合うフリーセルを探して，そのセルに割付けを行う．要求サイズを以上のフリーセルが見つからない場合は，スレッドローカル GC を発生させる．

オブジェクトを共有ヒープに割り付けるか，LOS に割り付けるかはバイトコード命令[7] 単位で決定する．オリジナルの Java バイトコード命令では，オブジェクトを割り当てるアロケート命令は `new`，`newarray`，`anewarray`，`multinewarray` の 4 つである．最適化ルーチンは，プログラム中の各アロケート命令ごとに，LOS に割り付けるか，共有ヒープに割り付けるかを最適化していく．

2.3　ライトバリア

Java バイトコードのうち，`putfield`，`putstatic`，`aastore` の 3 つの命令は共有ヒープ空間にオブジェクトの参照を書き込む命令である．これらの命令が参照を書き込む前に，書き込まれる参照と書き込むアドレスのチェックを行う．この処理を GC の流儀に従いライトバリアと呼ぶことにする．

TLH のライトバリア処理は，書き込みむ参照が LOS 上のものであれば，参照の指すオブジェクトに投機失敗をマークする．これは，オブジェクトのヘッダの中に 1 ビットの投機失敗フラグを用意することで，実現できる．

例をあげて説明する．プログラム内で

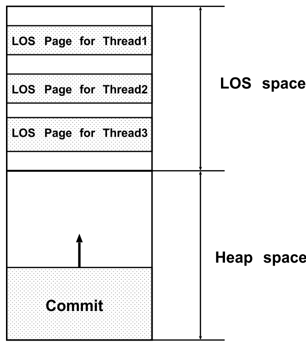

図 2 共有ヒープ空間と LOS 空間の配置
Fig. 2 Memory organization of Java heap with LOS spaces.

```
putfield( Object receiver,
          Field field,
          Object value ) {
  if( value >= SHAREDHEAP_LOS_BOUNDARY ){
    value->header_word |= MISSPECULATED_MARKED;
  }
  receiver[ field_offset(field) ] = value;
}
```

図 3 putfield 命令のライトバリア操作
Fig. 3 Write barrier code for putfield operation.

```
receiver.field = value;
```

の形で現れた `putfield` 命令があると，`value` で指されるオブジェクトがライトバリアの対象となる．まず，`value` が null でないかを判断し，次に `value` が LOS 上のオブジェクトがどうか調べる．もし，LOS 上のオブジェクトであればオブジェクト `value` に投機失敗のマークを打つ．

投機失敗フラグは，オブジェクト生成時に初期化され，ライトバリア処理によってセットされる．いったん投機失敗フラグがセットされると，リセットされることはない．そのため複数スレッドが同時にフラグ操作を行っても一貫性は崩れないので，アトミック操作は不要である．

ライトバリア処理を容易にするため，ランタイムのメモリ空間を**図 2** のように配置する．このメモリ配置を採用する場合，ライトバリア操作の null 判定と LOS 内かどうかの判定が，分岐命令 1 つで可能になる（**図 3**）．

2.4 スレッドローカル GC

LOS 上にあるオブジェクトは，投機失敗オブジェクトとそれ以外に二分される．また，投機に失敗していないオブジェクトは Java スレッドのスタックから到達可能なオブジェクトと，到達不能なオブジェクトに二分できる．到達可能なオブジェクトはスタック直接参照オブジェクトであり，到達不能なオブジェクトはガーベージである．

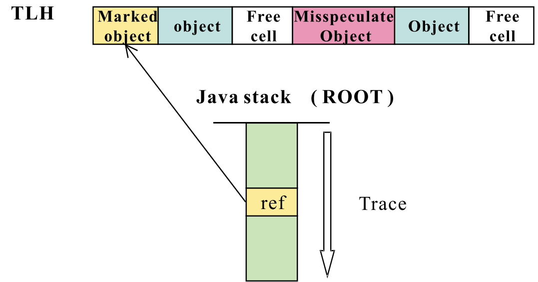

図 4 スレッドローカル GC: Marking Phase
Fig. 4 Thread Local GC: Marking Phase.

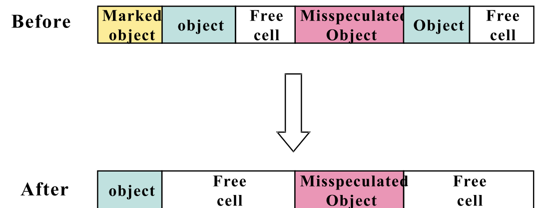

図 5 スレッドローカル GC: Sweeping Phase
Fig. 5 Thread Local GC: Sweeping Phase.

スレッドローカル GC は次の 2 つのフェーズにより LOS 上のガーベージを回収する．

Phase 1. marking phase

LOS 上のオブジェクトのうち，スタック直接参照オブジェクトにマークを行う．

このフェーズは，Java スレッドのスタックを走査し，LOS を指す参照を探す．スタックから指された LOS 上のオブジェクトはマークする．走査が終了するとスタック直接参照オブジェクトにのみマークがついている（**図 4**）．また，マークをされてないオブジェクトのうち投機失敗オブジェクト以外はガーベージである．

Phase 2. sweeping phase

LOS を走査してガーベージを回収し，フリーリストを再構築する．そのとき，連続するフリーセルは連結し，次回のスレッドローカル GC のために，スタック直接参照オブジェクトのマークはすべてリセットする（**図 5**）．

2.5 LOS リカバリ GC

実行が進むと LOS 中に投機失敗オブジェクトが増加していくため，LOS のフラグメンテーション化が進んだり，空き領域が消費し尽くされたりする．このような場合，スレッドローカル GC を起こしても要求とされるサイズの空き領域が確保できない．

そこで，すべての Java スレッドの一斉停止を行っ

た後，すべての LOS 上のオブジェクトのうち，生きているオブジェクトを共有ヒープに移動して，LOS を初期状態に戻す．この処理を LOS リカバリ GC と呼ぶ．

ここでは紙面の都合により LOS リカバリ GC の詳細は述べない．

LOS リカバリ GC が発生すると，一斉停止型 GC によるポーズ時間が伸びて実行性能が低下する．また，1 回の LOS リカバリ GC が長時間にわたる場合には，実時間性を損う可能性がある．そのため，TLH にとって LOS リカバリ GC の発生はペナルティとなる．

2.6 最 適 化

TLH はすべてのオブジェクトを LOS に割り付けることが可能である．しかし，投機失敗オブジェクトは LOS 中にフラグメンテーションを作り実行性能を落とす要因となる．また，投機失敗オブジェクトによって LOS リカバリ GC が頻発すると，一括停止型 GC の発生を抑えるという TLH の目的が失われてしまう．そのため，投機に失敗するオブジェクトは，最初から共有ヒープ割付けにする方が効率が良い．

投機に成功するか失敗するかは，主にアロケート命令が呼ばれたコンテキストに依存する．しかし，アロケート命令は，生成するオブジェクトの投機成功・投機失敗で偏った傾向を持つものが多い．そのため，各アロケート命令ごとに LOS 割付けにするか，共有ヒープ割付けにするかを決める最適化が可能になる．

以下，アロケート命令を LOS 割付けにするか共有ヒープ割付けにするかの判断を割付け戦略と呼ぶことにする．TLH では，割付け戦略の最適化として以下の 2 種類の手法を用いる．

2.6.1 静的コード解析による最適化

バイトコードのデータフロー解析によって，あるアロケート命令によって生成されたオブジェクトがどのようなパスを通っても `putfield`，`putstatic`，`aastore` 命令の代入値となるなら，このようなアロケート命令から生じるオブジェクトは必ず投機失敗する．このようなアロケート命令はあらかじめ LOS 割付けを禁止し，共有ヒープ割付けに最適化すべきである．

下のプログラムの場合，

```
L1: T object = new T();
L2: receiver.field = object;
```

L2 の `putfield` 命令で `object` はライトバリアされる．そのため，L1 の `new` が LOS 割付けの場合に，生成したオブジェクトが必ず投機に失敗することが分かる．そこで，L1 の `new` 命令は共有メモリ割付けに決定する．

また，L1 が共有メモリ割付けに最適化されると，L2 の `putfield` 命令は必ず共有メモリ割付けオブジェクトの参照を代入することになる．そこで，L2 の `putfield` 命令はライトバリア処理を省略することが可能になる．

このような解析はポインタ解析の一種である．解析の範囲を広げると，共有ヒープ割付けと決定できるアロケート命令は増えていくが，コストは大きくなる．解析を基本ブロック内に限定すればバイトコードの大きさ n に対して解析のコストは $O(n)$ に抑えることができる．

2.6.2 履歴情報を用いた最適化

静的コード解析によって，割付け戦略が決定できないアロケート命令は，投機成功・失敗の履歴情報を用いて最適化する．各アロケート命令ごとに，一定回数の履歴を収集して，投機失敗が多い場合は共有ヒープ割付けに，投機成功が多い場合は LOS 割付けに割付け戦略を決定する．

履歴情報の収集のために，メソッドごとにアロケート命令数分のエントリを持った履歴テーブルを用意する．履歴テーブルの各エントリには，該当するアロケート命令の位置情報，割付けを行ったオブジェクトの全個数，スレッドローカル GC によって回収できたオブジェクトの個数を記録する欄を設ける．

また，LOS 上のオブジェクトには，自分がどのアロケート命令で生成されたかを記録するために，1 ワード余分な領域を付加する．このワードに，履歴テーブルエントリへのポインタを書き込むことで，投機成功・失敗の履歴情報を更新することができる．

アロケート命令の割付け戦略の決定は，一斉停止型 GC の中で行う．履歴テーブルから一定回数以上オブジェクトを生成しているアロケート命令を撰択し，投機失敗率 fp を計算する．

N_{all}　全オブジェクト数
N_{lgb}　スレッドローカル GC 回収数

$$fp = \frac{(N_{all} - N_{lgb})}{N_{all}}$$

fp が一定値未満のアロケート命令は LOS 割付け，それ以上のものを共有ヒープ割付けと決定する．

履歴情報をどこまで集めてから割付け戦略を決定するか，投機失敗率の判定の基準をどこにおくかはヒューリスティックスによって与える．

3. 実　　装

我々は，TLH によって LOS に割り当て，スレッドローカル GC によって回収可能なオブジェクト（以降，LOS 割付け適合オブジェクト）の量を評価するためのシミュレータと，Sun Hotspot VM 1.4.0[1] をベースにした Java ランタイムの 2 つを開発した．

3.1 トレースベースシミュレータ

LOS 割付け適合オブジェクトの論理的な量を予備評価するために，トレースベースのシミュレータを作成した．このシミュレータは SUN JDK 1.3 の Classic VM をベースとしている．

シミュレーションは，オブジェクトのアロケート，メモリアクセスなどの実行履歴をすべて記録し，プログラム終了後に計算を行うことによって LOS 割付け適合オブジェクト量の理論値を算出する．

シミュレータには，2.6 節で述べた 2 種類の最適化機構も実装している．ただし実行時情報を用いた最適化は，1 度プログラムが完了した後の完全な実行履歴を用いている．またシミュレータは，実行中に履歴情報を保存していくため，実機の 1/1,000 程度の実行速度となる．そのため，タイマ割込みなどを使用したベンチマークなどは意味のあるデータがとれないことがある．

3.2 Sun Hotspot VM への TLH の実装

Sun Hotspot VM 1.4.0 上に TLH の実装を行った．

オリジナルの Hotspot VM では，オブジェクトをすべて共有ヒープ上に割り当てる．また世代別 GC を採用し，共有ヒープを新世代と旧世代の 2 つの世代に分割している．GC は新世代のみを回収する New GC と，すべての世代を回収する Full GC の 2 種類を発生させる．New GC ではコピー方式が使用され，Full GC ではマーク & コンパクト方式が使用される．

本論文では，TLH の実装のために Hotspot VM にいくつかの修正を行った．

- 新世代よりも若い世代3つ目の世代を設けた．この世代に各スレッドの LOS 用の領域を割り当てる．
- New GC，Full GC と同様の一斉停止型 GC として LOS リカバリ GC を実装した．LOS リカバリ GC は，任意のスレッドがスレッドローカル GC 後の LOS 割付けに失敗したときに引き起こされる．LOS リカバリ GC は，LOS 内の生きているオブジェクトをすべて新世代へ移動させる．
- スレッド固有メモリ領域へのオブジェクト割付け命令を実装するために，ランタイムの内部でだけ使用される TLH 用バイトコードを追加した．追加した命令はアロケート命令とヒープへの参照書き込み命令である．アロケート命令は `new`，`newarray`，`anewarray`，`multinewarray` のそれぞれに，履歴情報なしの LOS 割付け版と履歴情報付き LOS 割付け版を用意し，8 命令を追加した．参照書き込み命令は `putfield`，`putstatic`，`aastore` のそれぞれに，ライトバリア処理付き版を用意し，3 命令を追加した．拡張した命令はバイトコードの未使用オペコード領域に割り当てた．

また，現実的な Java 環境で GC の性能の評価を行うためには，JIT コンパイラを含めた実行性能を測ることが不可欠である．そのため Hotspot VM の持つ 2 種類の JIT コンパイラのうち，エンタープライズ分野のアプリケーションでより有効な Server JIT コンパイラを TLH に対応させた[8]．

3.3 TLH 最適化機構の実装

静的コード解析ルーチンは，ランタイムがクラスをロードした直後に解析を行い，バイトコードの書き換えを行う．共有ヒープ割付けに適するアロケート命令以外を履歴情報付き LOS アロケート命令に書き直す．また，`putifeld` 命令など参照書き込み命令を必要に応じてライトバリア付きの命令に置換する．同時に履歴情報テーブルを作成し，ランタイム内でメソッドを表すデータ構造にリンクする．

履歴情報を用いた最適化ルーチンは，各アロケート命令ごとに 128 個オブジェクトが回収されるまで履歴情報を集め，投機失敗率が 1/4〜1/16 未満の場合に LOS 割付け，それ以上だと共有ヒープ割付けというヒューリスティックスを採用した．

4. 評　　価

TLH の評価は，LOS 割付け適合オブジェクト量と，エンタープライズ分野のアプリケーションでの性能向上率の 2 点に着目して行った．

まず，LOS 割付け適合オブジェクト量を SPECjvm 98 ベンチマーク[9] を用いて評価した．SPECjvm98 は 7 本の小規模なプログラムからなるベンチマークスイートである．4.1 節では，完全な実行履歴を持っているという前提で割付け戦略を最適化した場合の LOS 割付け適合オブジェクトを，トレースベースシミュレータで評価した．これは，LOS 割付け適合オブジェクト量の論理的な上限となる．次に 4.2 節では，TLH を実装した Hotspot VM 上で，実際の LOS 割付け適合オブジェクト量を計測した．

エンタープライズ分野のアプリケーションでの性能の評価には，SPECjbb2000 ベンチマーク[10] を使用し

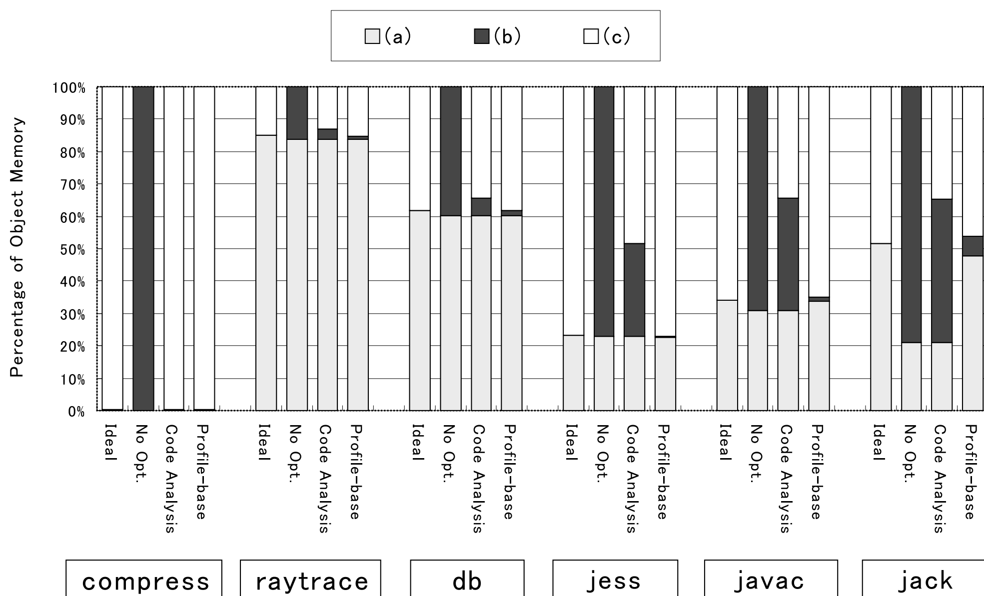

図 6 シミュレータによって割り付けられるオブジェクト量の割合（SPECjvm98）
Fig. 6 Distribution of object types in LOS spaces (SPECjvm98).

た．SPECjbb2000 は TPC-C ベンチマークを模した Java ベンチマークで，複数のスレッドが起動し，ベンチマーク内の擬似的なデータベースとの間でトランザクション処理を行う．そのため，エンタープライズ分野のアプリケーションの性質を模しているといえる．SPECjbb2000 において，トランザクションを処理するスレッドは warehouse（W.H.）と呼ばれる．また，SPECjbb2000 のスコアは単位時間に処理できたトランザクション数であり，数値が大きいほど性能が高い．

4.3 節では，複数のアーキテクチャ上で SPECjbb 2000 を計測した．4.4 節では，TLH のスケーラビリティを評価するために，16 ウェイの SPARC GP64 300 MHz マシンで CPU 数を変化させながら計測を行った．SPECjbb2000 の評価では，Server JIT コンパイラを有効にしている．

4.1 論理的な LOS 割付け適合オブジェクト量

3.1 節のトレースベースシミュレータによって，LOS 割付け適合オブジェクトが論理的にどの程度あるのかを評価した．このシミュレーションでは，オブジェクトを以下の項目に従って分類し，各オブジェクトのサイズの累積を計測した．

(a) LOS 領域に割り付けて，投機に成功したオブジェクト

(b) LOS 領域に割り付けたが，投機に失敗したオブジェクト

(c) 最初から共有ヒープに割り当てたオブジェクト

(a) は LOS に割り付けることによって，利得を得ることができる部分で，この部分が多いほどメモリ効率が上がる．(b) は逆に LOS 割付けによってペナルティとなる部分である．共有ヒープに割り付けることができなかった部分である．

SPECjvm98 の中から 6 つのベンチマークを以下の 4 つの方式について比較した．

Ideal アロケート単位で投機の成功・失敗が，完全に予測できたと仮定した場合の理論値．投機に成功するものは (a) に，投機に失敗するものは (c) としたデータ

No Opt. 最適化を行わず，すべてのオブジェクトを投機 LOS 割付けにした場合のデータ

Code Analysis バイトコードのローカル解析によって，最適化を行ったデータ

Profiled-base バイトコードのローカル解析に加えて，実行時履歴情報を用いた最適化を行ったデータ

シミュレーションの結果を**図 6** に示す．

このグラフから，TLH はベンチマークが使用するメモリ量の 20〜80%を LOS に割り当てて，スレッドローカル GC で回収することができると判断できる．

投機失敗オブジェクトの量は，jack で 6%になったが，それ以外のベンチマークでは 3%以下であることが分かった．特に jess，javac，jack のように Code Analysis を適用した段階で，投機失敗オブジェクトがほとんど削減できるものでも，Profiled-base を適用

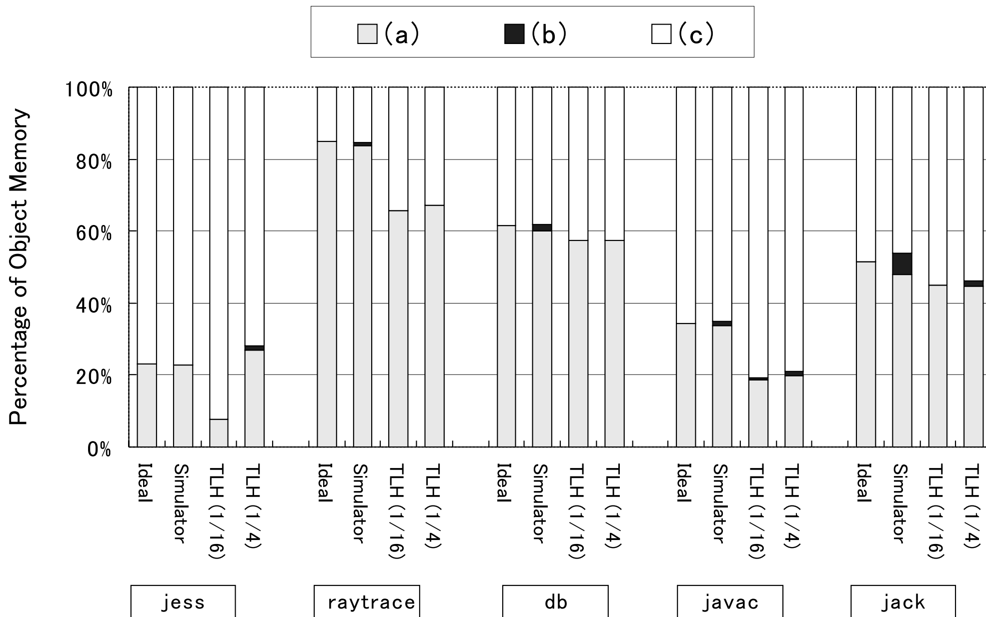

図 7 TLH によって割り付けられるオブジェクト量の割合（SPECjvm98）
Fig. 7 Distribution of object types allocated by TLH in LOS spaces (SPECjvm98).

表 1 領域別の割付けメモリ量の比（SPECjvm98）
Table 1 Ratio of allocated memory size among each area.

	jess		db		mtrt		javac		jack	
履歴情報最適化の方針	1/16	1/4	1/16	1/4	1/16	1/4	1/16	1/4	1/16	1/4
New GC 発生回数	1,684	1,311	102	207	383	366	1,107	1,098	649	644
Full GC 発生回数	2	42	5	10	19	19	98	106	8	24
LOS リカバリ GC 発生回数	14	20	13	23	10	10	148	18	11	17
スレッドローカル GC 発生回数	5,475	1,448	2,694	5,435	38,915	38,764	5,194	1,355	29,340	2,015
共有ヒープ割付け比率（%）	92.19	71.83	42.46	42.68	34.32	32.87	80.89	79.11	55.05	54.59
LOS 割付け比率（%）（投機成功）	7.78	26.98	57.48	57.28	65.59	67.04	18.61	19.83	44.92	45.41
LOS 割付け比率（%）（投機失敗）	0.03	1.19	0.06	0.04	0.09	0.09	0.50	1.06	0.03	1.33

した段階でほとんど削減でき，静的な最適化と動的な最適化を組み合わせた効果は大きいといえる．

4.2 LOS 割付け適合オブジェクト量

Hotspot VM に実際に TLH を実装した版で，LOS 割付け適合オブジェクト量の比較を行った．評価に用いたベンチマークは，SPECjvm98 のうち 4.1 節で計測したものである．ただし，compress は除外した．

パラメータはヒープの初期サイズを 16 M バイト，最大サイズを 32 M，スレッドあたりの LOS のサイズを 32 K とした．各ベンチマークを 10 回連続で実行した結果を**図 7** と**表 1** に示す．

図 7 は，4.1 節と同様に割付けオブジェクトの種類を分類したものである．Ideal は 4.1 節の Ideal と等しいが，Simulator は 4.1 節のグラフの Profiled-base にあたる．TLH(1/16) と TLH(1/4) は Hotspot VM ベースの TLH の測定結果である．

TLH(1/16) と TLH(1/4) の違いは，2.6.2 項で述べた履歴情報を用いた最適化の評価基準の違いである．TLH(1/16) は投機失敗率が 1/16 未満のアロケート命令を，TLH(1/4) は 1/4 未満のものを LOS 割付けに決定する．

表 1 は，各領域に割り付けられたオブジェクト量と，GC 回数を示す．表中の New GC，Full GC，LOS リカバリ GC，LOS リカバリ GC は GC 発生回数である．スレッドローカル GC はすべての Java スレッドで起こった回数をまとめている．

シミュレーションによる論理的なメモリ割付け量と比較すると，javac では 34%あった理論値が 18〜19%に，raytrace では 84%が 65〜67%に減少した．

シミュレーションと実機で異なるのは，シミュレーションでは完全な履歴情報を使って最適化を行っているが，実機では履歴情報の収集と最適化を実行時に並

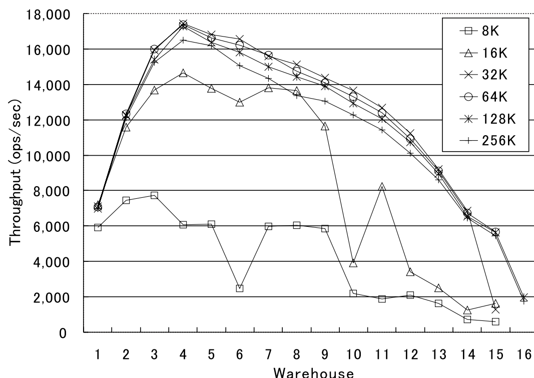

図 8 LOS サイズを変更した実行性能
Fig. 8 Performance comparison of LOS size.

行して行っている点である．その結果，javac や ray-trace では割付け戦略の誤りが増え，理論値よりも割付け量が低下している．

jess では，TLH(1/16) と TLH(1/4) で割付け量に差が出ている．LOS 割付け適合オブジェクト量が TLH(1/16) の 7.78%から TLH(1/4) の 26.98%に増加している．しかし，投機失敗オブジェクト量も 0.03%から 1.19%に増加している．

4.3 実行性能

SPECjbb2000 を用いた TLH の実行性能の測定を行った．

JavaVM のヒープサイズは初期値・最大値とも 288 M バイトとした．このサイズは以下の理由により決定した．SPECjbb2000 は基礎となるデータベース部分のために約 11 M バイト，W.H. ごとに 16 M バイトのメモリが固定的に必要となる．そのため SPECjbb2000 は W.H. 数が増えるに従って必要なメモリが多くなっていく．本論文では 1〜16 W.H. での性能測定を行うため，16 W.H. の実行に最低限必要なメモリサイズとして 288 M バイトを設定した．

最初に，Pentium3 1 GHz × 4 CPU において，LOS サイズを変えながら性能を測定した結果が**図 8** となる．

LOS サイズは TLH の実行性能を決定する，大きな要因の 1 つである．LOS サイズが大きい場合，スレッドローカル GC がおこる頻度が減少するが，フリーリストが長くなるため操作コストが増大する．LOS サイズが小さいときはその逆となる．また，LOS サイズによってキャッシュの挙動が変わる．サイズが大きすぎる場合キャッシュミス率が増大し，性能低下を招くことになる．

LOS のサイズは 32 K バイトでピーク性能を得た．64 K バイトも 32 K バイトと，ほぼ同じパフォーマンスとなる．しかし，128 K バイト，256 K バイトと

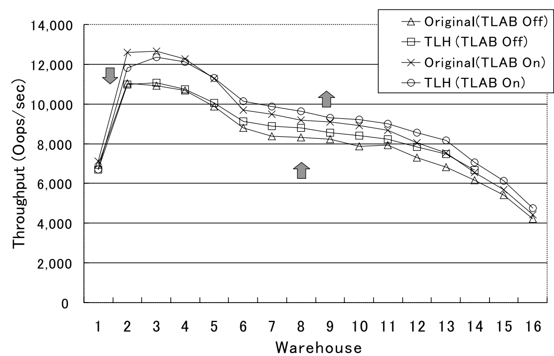

図 9 UltraSPARC III 700 MHz × 2 CPU
Fig. 9 UltraSPARC III 700 MHz 2 processors.

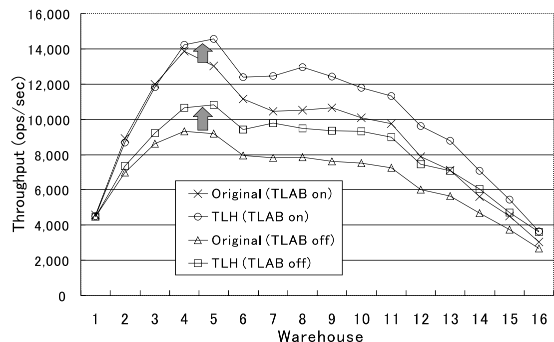

図 10 SPARC GP64 300 MHz × 16 CPU
Fig. 10 SPARC GP64 300 MHz 16 processors.

LOS の大きさが大きくなると，性能が低下していく．

逆に 8 K バイト，16 K バイトでは性能が大幅に落ちるうえに，W.H. ごとにスループットが変動し不安定なことが分かる．LOS サイズが小さ過ぎると，トランザクション 1 つ分のデータを保持するだけの容量がないためだと考えられる．

紙面の都合により省略するが，同様の結果が異なったアーキテクチャにおいても得られた．そこで，以下の性能測定では LOS サイズを 32 K バイトに固定している．

次に，Pentium3，UltraSPARC III，SPARC GP64 の 3 つのアーキテクチャで TLH の性能を測定した．この測定では，TLH とオリジナルの 2 つの Java ランタイムを，それぞれ TLAB を有効にした場合と無効にした場合の 4 方式の計測を行った．TLAB とは，オリジナルの Hotspot VM に備わるスレッド別メモリアロケータである．TLH と TLAB を併用する場合，LOS 割付けされるオブジェクトは TLH で管理され，共有ヒープ割付けされるオブジェクトは TLAB で管理することになる．

測定結果を**図 9**，**図 10**，**図 11**，**図 12** に示す．グラフには，TLH がオリジナルに比べて性能が向上し

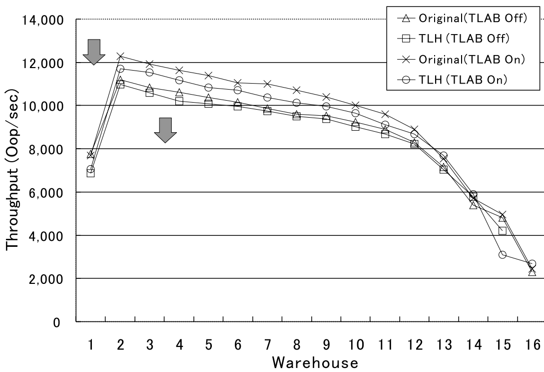

図 11 Pentium3 1 GHz × 2 CPU
Fig. 11 Pentium3 1 GHz 2 processors.

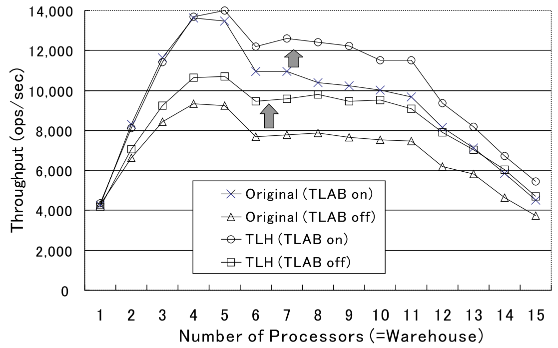

図 13 SPARC GP64 300 MHz CPU 数 1〜16 (SPECjbb2000)
Fig. 13 SPARC GP64 300 MHz from 1 to 16 processors (SPECjbb2000).

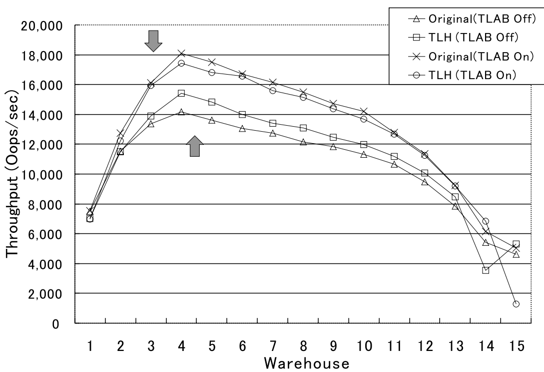

図 12 Pentium3 1 GHz × 4 CPU
Fig. 12 Pentium3 1 GHz 4 processors.

表 2 領域別の割付けメモリ量の比（SPECjbb2000）
Table 2 Distribution of object size allocated in the individual spaces.

共有ヒープ割付け	78.06%
LOS 割付け（投機成功）	21.94%
LOS 割付け（投機失敗）	0.00%

ている箇所では上向きの矢印，性能が低下しているところには下向きの矢印を入れて強調している．

表 2 は，LOS 割付け適合オブジェクト量の比であり，約 22%になった．この値はアーキテクチャや W.H. 数にはほとんど依存しなかった．

LOS 割付け可能オブジェクト量の比はアーキテクチャや W.H. 数の影響を受けないと考えられるので，表 2 の結果から SPECjbb2000 は 22%のオブジェクトを LOS に割り付け，投機に成功していると判断できる．

UltraSPARC III，SPARC GP64 では，TLH の方がピーク以降の W.H. でスループットの減少の仕方が緩やかである．ただし，ピーク性能の向上は無視できる範囲におさまっている．

Pentium3 では，TLH のスループット曲線がオリジナルのものよりも低く，全般に性能が落ちていることが分かる．ただし，CPU 数が増えるに従って差は縮まっている．

4.4 スケーラビリティ

SPARC GP64 300 MHz 16 CPU のマシンで，CPU 数を変えながら測定を行い，スケーラビリティを計測した．各 CPU 数の測定値は，CPU 数と等しくなる W.H. 数での計測結果である．

結果を**図 13** に示す．

ピーク性能は TLAB を無効にした場合には 14%ほど上がる．しかし，TLAB を有効にした場合には 2.8%しか上がらない．効果としてはわずかである．

しかし，ピーク以降の性能低下率は，オリジナルと比べて TLH の方が緩やかである．W.H. = 11 で TLAB を有効にした場合のピークからの性能低下率はオリジナルでは 29%だが，TLH では 18%になっている．

4.5 考　　察

SPECjbb2000 では，TLAB を無効にしたオリジナル Hotspot VM に対しては性能の向上を得た．しかし，TLAB を有効にした場合を比較すると，2〜4 CPU のマシンでは性能が低下している．この原因を考察してみる．

一斉停止型 GC を採用する Java ランタイムの場合，GC によってポーズしている時間とそれ以外の時間に大別できる．前者を GC ポーズ時間，後者をクライアント時間と呼ぶ（TLH の場合には，スレッドローカル GC に掛かる時間はクライアント時間に含める）．クライアント時間の総和と GC ポーズ時間の総和の比は**図 14** となる．このグラフは実行時間に占めるクライアント時間の割合を表している．グラフの 100%は

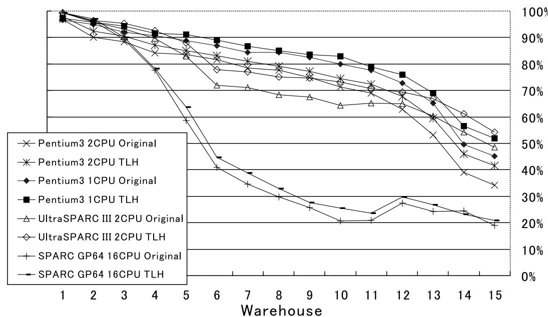

図 14 SPECjbb2000 計測中のクライアント時間の比（TLAB 有効）
Fig. 14 Ratio of client time in SPECjbb200 run.

一斉停止型 GC がまったく起きない状態である．

TLH のスレッドローカル GC は，共有ヒープ GC によって回収すべきオブジェクトを個別のスレッドで回収可能にする．つまり GC ポーズ時間の一部をクライアント時間に組み入れることになる．共有ヒープを対象とした一斉停止型 GC が単一の GC スレッドによって実行されるのに対して，クライアント側に組み入れられたスレッドローカル GC は CPU 数分の並列度で実行されるため，速度の向上を得ることができる．この結果，TLH は図 14 が示すように GC ポーズ時間を短縮する効果が得られる．

ただし，TLH にはスレッドローカル GC・ライトバリア・LOS の管理などの余分なオーバヘッドが必要となるため，クライアント時間あたりの実行性能が低下する．GC ポーズ時間の短縮効果とクライアント時間のオーバヘッドのバランスによっては，かえって実行性能が低下する．2〜4 CPU のマシンの TLAB を有効にした場合の計測では，オーバヘッドを上回る GC ポーズ時間短縮効果を得ることができなかったと考えられる．

以降 4.5.1 項，4.5.1 項では，クライアント時間と GC ポーズ時間に分けて TLH の挙動を見ていく．

4.5.1 クライアント時間

SPECjbb2000 は計測時間内に処理できたトランザクション数を計測秒で割って結果とする．そのためトランザクションを処理しているのはクライアント時間のみなので，処理トランザクションをクライアント時間で割ると，GC の影響を無視した場合の性能評価を行うことができる．

スループットをクライアント時間で割ったものを SPARC GP64 300 MHz × 16 CPU，Pentium3 1 GHz × 2 CPU，Pentium3 1 GHz × 1 CPU，UltraSPARC III 1 GHz × 2 について計測した．

結果は**図 15**，**図 16** となる．

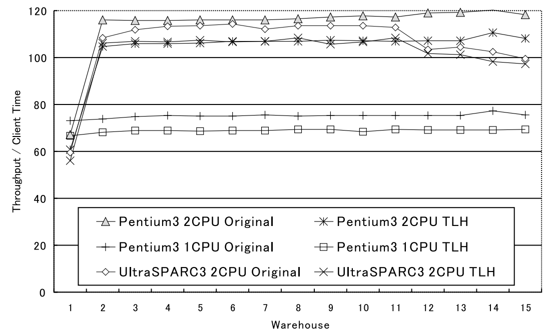

図 15 クライアント時間に対するスループット（TLAB 有効）（Pentium3 × 1 CPU，× 2 CPU，UltraSPARC III × 2 CPU）
Fig. 15 Throughput excluding GC pause time (TLAB enabled) (Pentium3 1 processor, 2 processors, UltraSPARC III 2 processors).

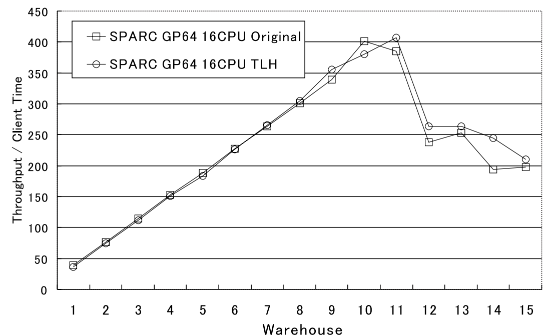

図 16 クライアント時間に対するスループット（TLAB 有効）（SPARC GP64 × 16 CPU）
Fig. 16 Throughput excluding GC pause time (TLAB enabled) (SPARC GP64 16 processors).

図 15 から，CPU が 2 個の UltraSPARC III，Pentium3 マシンでは，W.H. が 1 から 2 になったときにほとんど 2 倍の性能向上が得られていることが分かる．また，W.H. が 2 以降ではスループット比の低下が起きず，グラフの水平部分が続いている．

図 16 から，CPU が 16 個の SPARC GP64 マシンでは W.H. = 10 に届くまでは直線的に性能が伸び，その後は徐々に性能が低下することが分かる．

このことから，GC がまったく起こらない環境を仮定すると SPECjbb2000 は高いスケーラビリティが存在しているといえる．SPECjbb2000 のスケーラビリティを下げているのは，主に GC の影響であると特定できる．

また，図 15 の各グラフの水平部分を比較すると，TLH はオリジナルよりクライアント時間中の実行性能が低いことが分かる．この性能差は，TLH がクライアント時間中に掛かるオーバヘッドのコストであると考えられる．

オリジナルに対する TLH の性能ダウン率を計算

表 4 Pentium3 1 GHz × 2 CPU で SPECjbb2000 を実行した場合の GC の発生回数/総計時間/GC 1 回の時間（TLAB 有効）
Table 4 Characteristics of garbage collections in SPECjbb2000 run on Pentium3 1 GHz 2 processors (TLAB enabled).

W.H.	オリジナル						TLH								
	共有ヒープ GC（New GC）			共有ヒープ GC（Full GC）			共有ヒープ GC（New GC）			共有ヒープ GC（Full GC）			LOS リカバリ GC		
	回数（回）	時間（秒）	1 回（m 秒）	回数（回）	時間（秒）	1 回（m 秒）	回数（回）	時間（秒）	1 回（m 秒）	回数（回）	時間（秒）	1 回（m 秒）	回数（回）	時間（秒）	1 回（m 秒）
1	71	3.76	53.0	0	0.00		51	2.69	52.74	0	0.00		1	0.093	
2	114	10.74	94.2	1	1.08	1,075	87	9.44	108.54	0	0.00		0	0.000	
3	111	12.75	114.9	1	1.14	1,143	83	11.88	143.18	1	1.47	1,468	0	0.000	
4	104	14.95	143.8	3	4.00	1,331	79	13.13	166.14	2	2.84	1,422	0	0.000	
5	104	16.08	154.6	3	3.64	1,215	77	14.27	185.34	3	4.31	1,438	0	0.000	
6	100	16.60	166.0	4	5.31	1,328	74	15.57	210.42	4	5.40	1,351	0	0.000	
7	94	17.26	183.6	6	8.68	1,446	72	16.26	225.85	5	7.18	1,435	0	0.000	
8	93	18.20	195.7	6	8.76	1,459	69	16.41	237.83	6	8.70	1,450	0	0.000	
9	88	18.16	206.4	8	12.01	1,501	68	17.33	254.68	6	8.91	1,486	0	0.000	
10	82	17.62	214.9	11	16.75	1,522	60	16.65	277.27	9	14.86	1,651	0	0.000	
11	75	16.64	221.9	13	20.70	1,592	56	15.78	281.79	11	18.09	1,644	0	0.000	
12	63	14.77	234.4	18	29.73	1,652	48	14.41	300.29	12	25.00	2,084	0	0.000	
13	42	9.53	226.9	28	46.76	1,670	30	8.81	293.67	24	40.99	1,708	1	0.500	500
14	10	2.25	225.0	42	70.83	1,686	9	2.65	292.89	34	59.75	1,757	3	1.085	361
15	0	0.00		45	78.87	1,752	0	0.00		39	70.45	1,807	0	0.000	
16	0	0.00		27	49.24	1,823	0	0.00		19	35.73	1,880	1	1.158	1,158

表 3 クライアント時間中の各処理の割合（TLAB 有効）（SPECjbb2000; Pentium3 1 GHz × 2 CPU）
Table 3 Distribution of client times in SPECjbb2000 run (TLAB enabled) (Pentium3 1 GHz 2 processors).

実行	90.0%
オーバヘッド	10.0%
ライトバリアオーバヘッド	(4.6%)
フリーリスト操作オーバヘッド	(2.9%)
スレッドローカル GC オーバヘッド	(2.6%)

すると，Pentium3 1 GHz では 1 CPU のとき約 8%，2 CPU のとき約 10%，Ultra SPARC III では 2 CPU のとき約 8%である．SPARC GP64 × 16 CPU ではピークに達するまでのグラフはほぼ一致している．

最も性能低下率が大きい Pentium3 1 GHz × 2 を選び，オーバヘッドをさらに詳細に調べる．TLH 実行時にはオリジナルと比べてスループットの対動作時間比で，11%の性能低下が起きている．

TLH のオーバヘッドとしてはライトバリアのコスト，アロケート命令時のフリーリスト操作のコスト，Thread-local GC のコストの 3 種類が考えられる．それぞれを性能にあたえている影響を計測すると**表 3** のようになる．

全体の性能低下の半分程度をライトバリアのコストが占めている．Thread-local GC とフリーリスト操作のオーバヘッドがほぼ半々程度，性能低下の原因になっている．

このうちフリーリスト操作のオーバヘッドは，現在の最も単純な形のフリーリストの実装から，オブジェクトサイズによってフリーリストを使い分けるより高度なデータ構造に変更することで，削減できる可能性がある．

4.5.2 GC ポーズ時間

Pentium3 1 GHz × 2 CPU 上で SPECjbb2000 を計測すると，各 GC の発生回数，総和時間，1 回の時間は**表 4** となった．

この結果を見ると，TLH は共有ヒープに対する一斉停止型 GC の発生回数を減らす効果があることが分かる．W.H. 1〜16 までの全体で共有ヒープ GC 回数を 23.4%削減できている．

しかし，共有ヒープに対する一斉停止型 GC の総時間は，13.8%しか削減できていない（**図 17**）．

GC ポーズ時間の削減率が，GC 回数の削減率よりも悪くなる理由は，1 回あたりの GC ポーズ時間が New GC で 19%，Full GC で 4%ほど増加するためである．

1 回あたりの GC ポーズ時間が増える原因は，回収の対象となるオブジェクトの寿命の長さが異なるためだと考えられる．一般に GC を越えて生き残るオブジェクトの個数が多くなるほど，GC の処理時間は長くなる．そのため，寿命の短いオブジェクトを回収するよりも寿命が長いオブジェクトを回収する方が，時間コストが高い．今，LOS に割り当てられる 22%のオブジェクトと，共有ヒープに割り当てられる 78%の

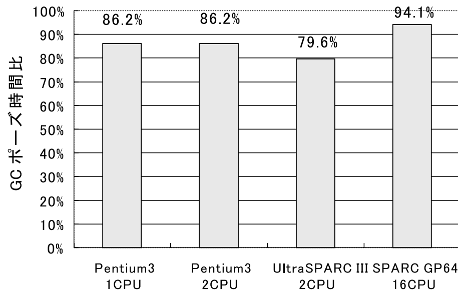

図 17 オリジナルに対する TLH の GC ポーズ時間比（TLAB 有効）

Fig. 17 Improvement of GC pause time for TLH compared to original VM.

オブジェクトでは，LOS に割り当てられるオブジェクトの方が寿命が短い．そのため，TLH によって一斉停止型 GC の回収対象から外せるオブジェクトは平均より寿命が短くなる．そのため，削減できるオブジェクト量に比例した GC ポーズ時間を削減効果は得られない．

4.6 LOS リカバリ GC の影響

2.4 節で述べたように，LOS リカバリ GC は TLH にとってペナルティとなる．

しかし，表 1 を見ると，SPECjvm98 を実行中には LOS リカバリ GC は New GC の 1 割以下程度しか発生していない．また，表 4 を見ると，SPECjbb2000 では最後の数 W.H. 以外では LOS リカバリ GC は発生していない．このデータより LOS リカバリ GC の起こる回数は通常の共有ヒープ GC 回数より十分に少ないため，スループットにあたえる影響は小さいと判断できる．

これは，2.6 節で述べた 2 つの最適化が有効にはたらき，スループットの低下を防いだためだといえる．

5. 関連研究

Doligez らの研究[2] は，初期のスレッドローカル GC の研究である．この研究では，ML 言語をベースとしてマルチスレッド機能を追加した Concurrent Caml Light を対象としている．スレッド固有メモリ領域に，値が更新されることのない不変オブジェクト（immutable object）を割り当て，スレッドローカル GC によって回収している．不変オブジェクトは言語の仕様上スレッドローカルであるで実行時にスレッドローカル性を検査する必要はない．

Steensgaard の研究[3] は，Java を対象としてエスケープ解析を用いてスレッドローカルオブジェクトを抽出する研究である．Steensgaard の研究の手法でスレッドローカル性を並列 GC アルゴリズムのために用いていて，スレッドローカル GC の可能性は示唆されているにとどまる．

Domani らの研究[11] は，本論文の手法と同様にランタイム支援による実行時チェックで投機失敗オブジェクトを検出している．ただし，Domani と本論文ではライトバリア手法が異なっている．Domani らの手法では同一のスレッド固有メモリ領域内のオブジェクトが互いに参照しあうことを許している．そのため，本論文よりも広い範囲のオブジェクトを，スレッドローカル GC の回収対象とできる．

しかし，Domani らの手法はライトバリア処理のオーバヘッドが大きい．Domani らの手法は以下のように手順でライトバリアを行うためである．(1) オブジェクトの参照 A が書き込まれる場合，A がスレッド固有領メモリ域のオブジェクトの参照かどうかを判定する．(2) A がスレッド固有メモリ領域を指している場合，A を書き込む領域が A の所属するスレッドのスレッド固有メモリ領域かどうかを判定する．(3) 書き込み先が A のスレッド固有領メモリ域以外の場合には，A は他のスレッドから参照可能となるので，投機失敗と判定されグローバルフラグをマークする．(4) A から到達可能な他のオブジェクトも再帰的にグローバルフラグをマークする．

本論文の手法と比較して，(2)，(4) の処理が余分に必要となる．Domani らはインタプリタモードでこのコストを計測し 1.5〜2.1%と評価している．そのため JIT コンパイラ上でのライトバリアのコストはさらに大きくなると予測できる．

6. ま と め

本論文では軽量なコード解析と実行履歴情報を用いた最適化により，スレッド別メモリ管理を実現する手法であるスレッドローカルヒープ（TLH）を提案した．

そして Sun JDK 1.3 Classic VM を基にして，TLH の動作を模擬したトレースベースのシミュレータを作成した．また，Sun Hotspot VM 1.4.0 上に TLH を実装し，さらに Server JIT コンパイラを TLH に対応させた．

シミュレーションを用いた評価より，TLH を用いることによって多くのベンチマークで全オブジェクト量の 20〜80%をスレッド固有メモリ領域に割り当て，スレッドローカル GC で回収可能であるという結果を得た．

SPECjbb2000 を実機で性能測定し，スレッド固有メモリ領域のサイズは Java スレッドあたり 32 K バイト程度が最もパフォーマンス高いことが分かった．また，

トランザクション処理中のオブジェクト量の 22%程度を，スレッドローカル GC で回収することが可能だった．投機失敗となるオブジェクトの量はきわめて少なくほとんど 0%と見なせ，最適化が有効にはたらいていることを確認できた．

オリジナルの Hotspot VM と比較した場合，8〜16 ウェイの大規模 SMP マシンにおけるスケーラービリティの評価では，ピーク性能が約 3%向上し，ピーク後の性能低下を大幅に防ぐことができた．

しかし，2〜4 way の SMP マシンではピーク実行性能がわずかに低下した．この規模の SMP マシンでは，並列にスレッドローカル GC を行うことによる利得よりもライトバリア処理などに掛かるオーバヘッドのペナルティが大きくなっただと考えられる．このような実行時チェックのオーバヘッドはクライアント時間の 8〜10%程度となった．

参考文献

1) Sun Microsystems, Inc.: Performance Documentation For The Java Hotspot Virtual Machine. http://java.sun.com/docs/hotspot/index.html
2) Doligez, D. and Xavier, L.: A concurrent generational garbage collector f or a multithreaded implementation of ML, *Proc. POPL'93*, pp.113–123 (1993).
3) Steensgaard, B.: Thread-Specific Heaps for Multi-Threaded Programs, *Proc. ISMM'00*, pp.18–24 (2000).
4) Choi, J., et al.: Escape Analysis for Java, *Proc. OOPSLA'99*, pp.1–19 (1999).
5) Bogda, J. and Hölzle, U.: Removing Unnecessary Synchronziation in Java, *Proc. OOPSLA'99*, pp.35–46 (1999).
6) Whaley, J. and Rinard, M.: Compositional Pointer and Escape Analysis for Java Programs, *Proc. OOPSLA'99*, pp.187–206 (1999).
7) Lindholm, T. and Yelin, F.: The Java^{TM} Virtual Machine Specification Second Edition. http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html
8) Paleczny, M., et al.: The Java HotSpot^{TM} Server Compiler, *Proc. Java^{TM} Virtual Machine Research and Technology Symposium* (*JVM*), USENIX Association (2001).
9) Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/
10) Standard Performance Evaluation Corporation: SPEC JBB2000 Benchmark. http://www.spec.org/osg/jbb2000/
11) Domani, T., et al.: Thread-Local Heaps for Java, *Proc. ISMM'02*, pp.76–87 (2002).

(平成 14 年 7 月 29 日受付)
(平成 15 年 1 月 9 日採録)

中村 実

昭和 50 年生．平成 11 年東京大学工学部電気工学科卒業．平成 13 年同大学院工学系研究科情報工学専攻修士課程修了．同年，株式会社富士通研究所に入社．現在に至る．言語処理系の研究に従事．

前田 宗則（正会員）

昭和 39 年生．平成元年大阪大学大学院基礎工学研究科物理系専攻修士課程修了．同年，富士通株式会社入社．平成 4 年〜平成 10 年技術研究組合新情報処理開発機構に出向．現在株式会社富士通研究所に所属．平成 15 年大阪大学大学院基礎工学研究科博士課程修了．工学博士．

小沢 年弘（正会員）

昭和 35 年生．昭和 57 年横浜国立大学工学部電気工学科卒業．昭和 59 年同大学院修士課程修了．同年，株式会社富士通研究所入社．言語処理系，計算機アーキテクチャの研究開発に従事．