システムソフトウェアと
オペレーティング・システム 78－4
（1998．5．7）

Java VM の GC の高速化

河場基行† 志村浩也† 木村康則†

JIT (Just In Time) コンパイル技術の進歩に伴い，Java 処理系が高速化が進んでいる．このため GC(ガベージコレクション) を含むメモリ管理が，全実行時間に占める割合が多くなってきている．Java 処理系の高速化は，GC の高速化が鍵となっている．

我々は，GC の高速化手法として 3 つの方式 (Allocation History GC, Hasty Compaction, Segregated Memory Management) を提案する．これら 3 方式の中でメモリコンパクションを取り除いた GC 方式である Segregated Memory Management はとくに有効であった．javac を実行した場合，総実行時間を 48.0%，GC に要する時間を 67.2% 短縮することが可能となった．

Fast Garbage Collection for Java VM

MOTOYUKI KAWABA,† KOUYA SHIMURA† and YASUNORI KIMURA†

For the sake of JIT(Just-In-Time) compiler, the Java virtual machine has worked faster. Garbage collecting becomes the critical part of Java virtual machine. It is significant to improve garbage collecting for total speedup.

In this paper, we present three strategies for garbage collection (Allocation History GC, Hasty compaction, Segregated memory management). We have designed and implemented the strategies on JDK1.2beta3.

The segregated memory management, which is a kind of GC without heap compaction, appears the best strategy among them. It can reduce 48.0% of total execution time and 67.2% of the cost of garbage collecting for the javac benchmark.

1. はじめに

Java 処理系は機種非依存なコード (Java バイトコード) を，JVM と呼ばれる仮想マシンで動作させる．このために Java はプラットフォームを選ばないプログラミング言語として急速に普及しつつあり，JVM の高速化は重要な技術となっている．

JVM の高速化は JIT(Just-In-Time) コンパイラによる命令解釈の高速化とメモリなどのリソース管理の高速化に分けられる．最近の JVM は，実行時に Java バイトコードからマシンコードに変換・実行する JIT コンパイル技術を使用しており，命令解釈に要する時間を大幅に短縮する工夫が施されている．このためにリソース管理に要する時間が実行時間に占める割合は大きくなりつつある．

Java のようなガベージコレクション機能を有した処理系では，とくにメモリ管理に要する時間が大きい．そこでメモリ管理の高速化に着目し，メモリアロケーション・ガーベジコレクションの方式を提案する．ここでは我々が開発した 3 手法 (Allocation History GC, Hasty Compaction, Segregated Memory Management) についての方式説明・評価を行なう．最近の JIT コンパイラの普及を鑑み，全て JIT コンパイラの使用を前提として評価を行なうこととする．

第 2 節で JVM の実行内訳，第 3 節で JVM のヒープ構造を示す．第 4 節で JVM の GC の特徴・および問題点を指摘する．第 5 節では 3 つのメモリ管理方式についての説明，第 6 節で性能比較結果を示す．

2. JVM の実行内訳

図 1 に JIT コンパイラを用いた場合の JVM (JDK1.2) の実行内訳を示す．(ベンチマークテストとしては javac を取り上げた．) 命令解釈・実行に要する部分は，「ハンドル参照」，「その他の実行」が相当する．これらが総実行時間に占める割合は 44.2%である．JIT コンパイラによる速度向上は「その他の実行」部分に含まれる命令解釈・実行部分にしか適用できない．したがって，JIT コンパイラによって解釈・実行を 2 倍高速化しても，全体の実行速度が高々30%程度しか向上しない．JVM の実行速度を向上させるには全実行時間の半分以上を占めるリソース管理の処理時間を短縮する必要がある．[1)]

† (株) 富士通研究所
FUJITSU LABORATORIES LTD

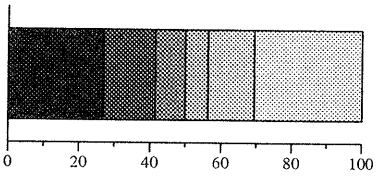

図 1 JVM の実行内訳

リソース管理の内訳をみると，メモリアロケーション・ガベージコレクションといったメモリ管理に 41.4% の時間を要していることがわかる．JVM を高速化するためには，特にメモリ管理を高速にすることが重要である．

3. JVM のメモリ管理方式

ガベージコレクションについて論議する前に,JVM のメモリ管理方式について簡単に説明する．JVM の処理対象となるオブジェクトは全てヒープに割り当てられる．JVM のヒープ構造を図 2 に示す.[1),2)] ヒープ領域はハンドル領域とオブジェクト領域に分割されている．通常オブジェクトへのアクセスはハンドルと呼ばれる構造体を介して行われる．

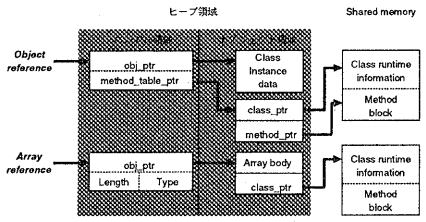

図 2 JDK1.2 のメモリ構造

参考文献[5)] の分類に従えば，Mark-Compact Collection 方式によって不要となったオブジェクトの回収を行う．Mark-Compact Collection 方式とは，Mark& Sweep によって不要なオブジェクトの回収を行った後，回収されなかったオブジェクトを圧縮する (移動・コピーする) オブジェクト回収方式である．JVM がハンドルを介してオブジェクトをアクセスする理由は，オブジェクトの移動・コピーを簡単にするためであると考えられる．図 3 にその様子を示す． 各ハンドルとインスタンスデータとの対応をうまく保存することで，インスタンスデータの中身を変更することなく，オブジェクトの移動を行うことができる．ハンドルとインスタンスデータは 1 対 1 対応になっているため，比較的容易に対応の保存ができる．

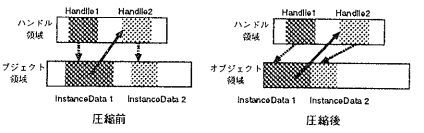

図 3 ハンドルを用いたオブジェクト移動

4. JVM の GC の特徴および問題

JVM のガベージコレクションの特徴およびその問題について述べる．

4.1 オブジェクトの生存期間

オブジェクトがアロケートされてから，ガベージコレクションによって回収されるまでの期間をオブジェクトの生存期間という．いま生存期間中に起動された GC の回数を，生存期間の長さと定義する．図 4 に生存期間の長さのヒストグラムを示す．

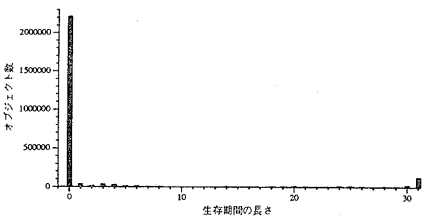

図 4 オブジェクトの生存期間

図 4 から生存期間の長さが 0 のオブジェクトが圧倒的に多いことがわかる．生存期間の長さが 0 のオブジェクトを低コスト (短時間) で回収できるのであれば，処理時間の長い GC の起動回数を減らすことができ，実行速度の向上が得られる．

4.2 GC の実行内訳

図 5 に GC の実行内訳を示す．JDK では，ガベージコレクション方式として，Mark-Compact Collection 方式を採用している．この方式は次の 4 つの処理フェーズから構成される．

Init ガベージコレクションに使用するテーブルを初期化をする．

Mark オブジェクトの参照木をトラバースし，マークをつける．

Sweep マークのつけられていないオブジェクトを回収する．

Compaction オブジェクトを移動することによってメモリコンパクションを行う．

図 5 より Init に要する時間がほとんどなく，Mark に処理時間の 70% 近くを費やしていることがわかる．

4.3 ヒープコンパクションの問題

java に **-verbosegc** オプションをつけて起動する

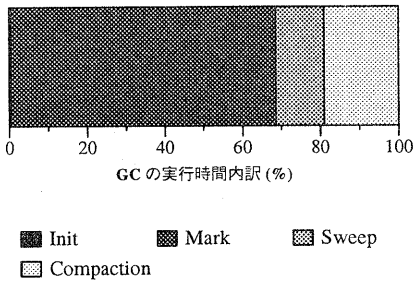

図 5 javac を実行したときの GC の実行時間内訳

と，ガベージコレクションの情報が出力される．(図 6 参照) 図 6 によると，ガベージコレクションで必ずしもヒープコンパクションが行われるわけではないことがわかる．

```
<GC: init&scan: 2 ms, ........, compact: 0 ms>
<GC: init&scan: 2 ms, ........, compact: 0 ms>
<GC: init&scan: 2 ms, ........, compact: 534 ms>
<GC: init&scan: 2 ms, ........, compact: 0 ms>
```

図 6 verbosegc 指定時の出力

コンパクションが行われているケースを見ると (たとえば図 6 の 3 番目の出力)，1 回の GC に要する時間が 864 ミリ秒であるのに対し，コンパクションに要する時間が 534 ミリ秒もかかっている．このことからヒープコンパクションは非常にコストの高い処理であることがわかる．JVM では GC に要する時間を短縮するために，コンパクション処理を行わないケースを設けているものだと推測できる．しかしこの方式はヒープフラグメンテーションを引き起こす可能性がある．

図 7 は javac を実行した場合のメモリ使用量の推移を示したものである． 総空きサイズが増えている箇

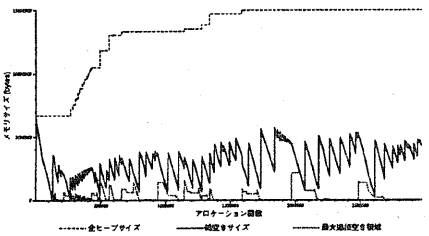

図 7 javac 実行時のメモリ使用量の推移

所は GC が起動された箇所である．また最大連続空きサイズが急激に増えている箇所はコンパクションが行われた箇所である．コンパクションを起動する直前は，総空きサイズと比較して最大連続空きサイズが非常に小さく，GC の起動回数が非常に多い．明らかにヒープフラグメンテーションが生じていることがわかる．

5. GC 方式の提案

第 4 節で説明した特徴・問題点をふまえて次の 3 つの GC 方式を提案する．

- Allocation History GC
- Hasty Compaction
- Segregated Memory Management

これらはいずれも Mark & Sweep 方式のガベージコレクションをベースにしている．以降それぞれの GC について説明を行う．

5.1 Allocation History GC

第 4.1 節で説明したように JVM のオブジェクトは比較的短期間に不要なオブジェクトとなる．Allocation History GC は短期間に不要となるオブジェクトの回収を高速に行うための手法である．とくに Allocation History GC では一時的に生成されるオブジェクト (他のオブジェクトからの参照が発生しないオブジェクト) を低コストで回収することを目的としている．一時的なオブジェクトが生成される例を示す．

```
String s = new String("a");
s.append(3);
```

このコードは図 8 に示すように一時的な String オブジェクトを生成する．図 8 の String オブジェクトは，他のオブジェクトから参照されることはない．

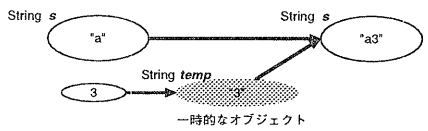

図 8 一時的オブジェクトの生成例

5.1.1 アルゴリズム

ここでは説明を簡単にするために，オブジェクトとハンドルを同一視して説明を行なう．アロケートされたオブジェクトを記録するアロケーション履歴バッファを用意する．オブジェクト回収の対象となるのはアロケーション履歴バッファに記録されたオブジェクトのみである．アロケーション履歴バッファがいっぱいになった時点で Allocation History GC が実行し，一時的オブジェクトを回収する．Allocation History GC 後にアロケーション履歴バッファを空にする．オブジェクトのアロケートができなくなった場合，Mark & Compaction GC を実行しオブジェクトを回収する．

Allocation History GC は，参照カウンタによる GC に似ている．より詳細な説明を以下に述べる．オブジェクトアロケート時に，有限長のアロケーション履歴バッファにオブジェクトへのポインタを登録する．これと同時にオブジェクトに対応した参照ビットテーブルのビットを 0 に初期化する．(図 9) アロケート

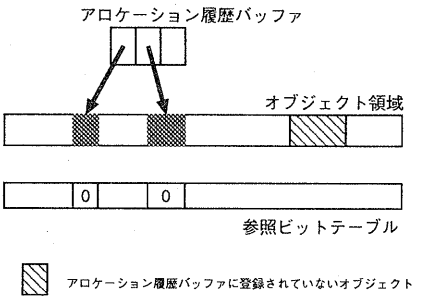

図 9 アロケーション履歴バッファへの登録

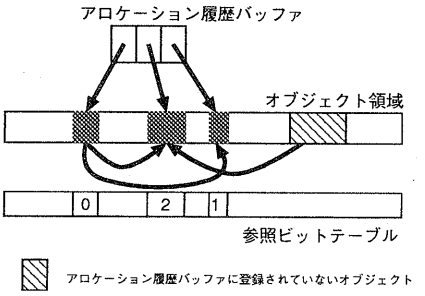

図 10 参照ビットのインクリメント

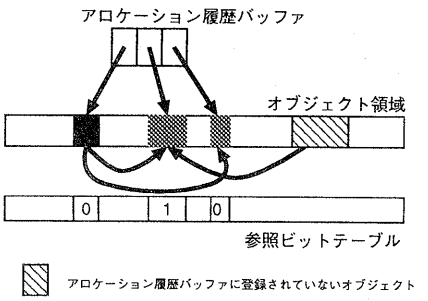

図 11 参照ビットのデクリメント

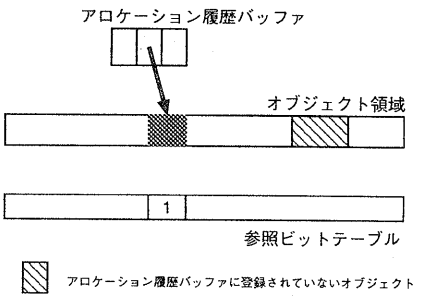

図 12 オブジェクト回収

したオブジェクトへの参照が発生したとき，参照ビットテーブルをインクリメントする．(図 10) アロケーション履歴バッファがいっぱいになった時点で，オブジェクトの回収を行う．

オブジェクト回収は以下の手順で行う．アロケーション履歴バッファに記録されているオブジェクトのうち参照ビットテーブルの値が 0 であるオブジェクトをルートノードとして参照木をたどりながら，参照ビットテーブルをデクリメントしていく．(図 11) つぎにアロケーション履歴バッファを走査し，参照ビットテーブルが 0 となったオブジェクトを解放する．(図 12) オブジェクトの参照木をたどる深さはオブジェクトに対して用意された参照テーブルのビット幅に制限される．我々がインプリメントした Allocation History GC では深さ 1 の走査しか行なわない．

参照ビットテーブルのデクリメントは Allocation History GC でしか行わないため，参照ビットテーブルの値は実際の参照数より大きな値になる．

5.2 Hasty Compaction

第 4.3 節で示したヒープフラグメンテーションの問題を解決するため，ヒープコンパクションを積極的におこなうことを考える．しかしヒープコンパクションが処理に時間がかかるものである以上，毎回ヒープコンパクションを行うわけにはいかない．メモリフラグメンテーションが生じているかどうかの判定を行ない，それに応じてヒープコンパクションを行うことが得策といえる．そこでヒープフラグメンテーションの指標として次式で定義されるフラグメンテーション度を導入する．

$$\text{フラグメンテーション度} = \frac{\text{総空きサイズ}}{\text{GC 起動時の要求サイズ}}$$

GC が起動された直後にフラグメンテーション度を評価する．GC が起動されたということは要求されたサイズの連続空き領域がないことを意味するから，フラグメンテーション度はヒープフラグメンテーションの指標となりうる．Hasty Compaction は上記のフラグメンテーション度に基づいて，ヒープコンパクションを行う方式である．フラグメンテーション度が一定の閾値をこえている場合，Mark& Sweep 後にヒープコンパクションを行う．

5.3 Segregated Memory Management

第 5.2 で述べたように，ヒープコンパクションは非常にコストの高い処理である．そこでヒープコンパクションを行わない GC を考える．ヒープコンパクションを行わない GC は以下のような長所・短所がある．

- 長所
  - コンパクション処理を行わないため，GC1 回当りの処理時間が短縮される．
  - ハンドル参照のオーバヘッドをなくす事ができる．(第 3 節参照)

- 短所
  - メモリフラグメンテーションのために，GC の起動回数・ヒープ使用量の増加がおきる．

Segregated Memory Management では，上記の長所を保ちつつ，短所の影響を小さくすること目的とする．ヒープコンパクションを行わない場合，全 GC 実行時間の 19%を占めるヒープコンパクションを排除することができると同時に，全実行時間の 13% を占めるハンドル参照を排除することが可能である．(図 5，図 1)

5.3.1 ヒープコンパクションなし GC

JDK1.2 の JVM から単純にヒープコンパクションを取り除いた時の，使用ヒープ量の推移を図 13 に示す．(ベンチマークは javac)

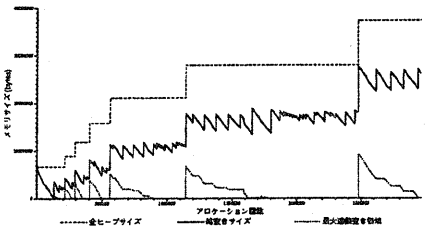

図 13 コンパクションなし GC のヒープ使用量の推移

ヒープコンパクションがある場合，使用ヒープ量が約 14MBytes 程度だったのに対し，ヒープコンパクションを行わない場合では，約 36MBytes のヒープ領域を消費する．アロケーション回数が増えるにつれて，総空きサイズが増えていくことからヒープフラグメンテーションが生じていると考えられる．

Segregated Memory Management では，オブジェクトをアロケートするときにオブジェクトサイズによってクラス分けを行う．さらに異なったクラスのオブジェクトを違うメモリブロックに割り当てることにより，ヒープフラグメンテーションによる使用ヒープ量増大を抑制する．以下にアルゴリズムの詳細を示す．

5.3.2 オブジェクトのアロケート

5.3.2.1 オブジェクトサイズのクラス分け

アロケート時にオブジェクトは以下の式にしたがってクラス分けされる．

$$A = \begin{cases} Min & (n \leq Min) \\ 2^{\lceil \log_2 n \rceil} & (Min < n \leq M) \\ \lceil n/M \rceil \times M & (n > M) \end{cases}$$

A クラス　n 要求されたサイズ
Min 最小クラス　M メモリブロックサイズ

Min より小さいオブジェクトは Min にアライメントし，M より大きいオブジェクトは M にアライメントされる．Min より大きく M より小さいオブジェクトは 2^n バイトにアライメントされる．我々の JVM では Min を 16bytes, M を 4Kbytes に設定している．

5.3.2.2 オブジェクト管理

ヒープは M バイトのメモリブロックに分割されている．同一メモリブロック内には同一クラスのオブジェクトしか割り当てない．ヒープ中の空き領域は次の 2 つのテーブルによって管理されている．

ブロック管理テーブル

各メモリブロックに毎に以下の情報を保持している．(図 14 にブロック管理テーブルのエントリを示す．)

- メモリブロックのクラス，メモリブロックのサイズ (オブジェクトサイズが M バイトを超える場合に必要)
- 最大空きチャンク数 (= メモリブロックのクラス/M)
- 空きチャンク数，空きチャンクのフリーリスト

空きブロックリスト

空き領域を持つメモリブロックをリスト構造で保持している．オブジェクトクラスによって別々のリストを持つ．(図 15)

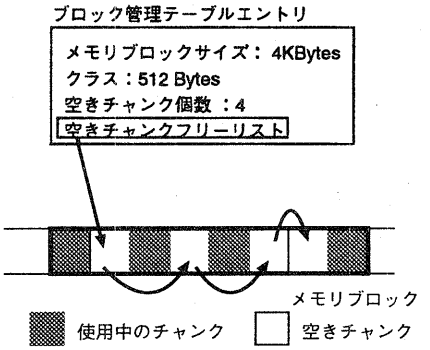

図 14 ブロック管理テーブル

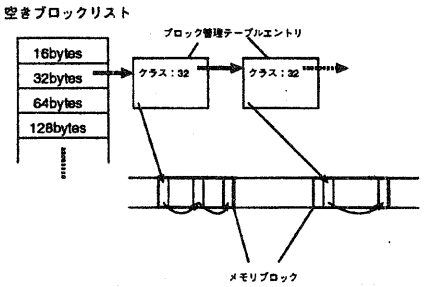

図 15 空きブロックリスト

5.3.2.3 オブジェクト割り当て

オブジェクトサイズによるクラス分けの後，空きブロックリストからチャンクを割り当てる．これと同時にブロック管理テーブルの情報 (空きチャンク個数，空きチャンクリスト) が更新される．次回のアロケート時間を短縮するために，空きチャンク数が 0 になったメモリブロックは，空きブロックリストから削除される．

もし所望のクラスの空きブロックが無かった場合，新しいメモリブロックを確保する．新しいメモリブロックには，空きメモリブロック (最大空きチャンク数と空きチャンク数が等しいメモリブロック) が割り当てられる．

オブジェクトのアロケート後に，ガベージコレクションの Mark フェーズ，Sweep フェーズを高速化するため，GC 用に用意されたビットマップ (Valid ビットマップ) にマークをつける．我々の JVM では Valid ビットマップは 16 バイト毎に 1 ビット用意している．

5.3.3 ガベージコレクション

我々の JVM は，JDK1.2 の JVM と同様 Mark& Sweep を採用している．Mark フェーズ，Sweep フェーズにおいて，Valid ビットマップを用いてオブジェクトの走査を行う点が異なる．この方式を採用した理由は次の通りである．

- Valid ビットマップを用いることで，複数のオブジェクトを同時に走査することができる．
- ハンドル参照を行なわない JVM ではハンドルがないために，オブジェクトを走査することが困難である．Valid ビットマップを用いることで，オブジェクトの所在を高速に判定することができる．

オブジェクトが解放されたとき，オブジェクトを含むメモリブロックのブロック管理テーブルが更新される．具体的には空きチャンク数をインクリメントし，オブジェクトを空きチャンクリストに挿入する処理を行なう．

5.3.4 空きメモリブロックのマージ

Mark & Sweep 後に全てのブロック管理テーブルを走査し，空きメモリブロックのマージを行なう．この走査は M バイト毎に割り当てられたブロック管理テーブルに対して行うため非常に高速である．

5.3.5 Handle の除去

ハンドルを除去した後のメモリ構造は[2] と同様なものとなった．図 16 にハンドルを除去した JVM のメモリ構造を示す．我々が開発した JIT コンパイラ[1] を変更し，ハンドルなし JVM の JIT をポーティングした．

6. 性能比較

JDK1.2beta3 をベースに，提案した 3 つの方式をインプリメントし性能比較を行なった．以下の実験結果は全て UltraSPARC II 250MHz 上で実行したものである．実行に際しては，我々が開発した JIT コンパイラ[1] を使用している．ベンチマークテストは javac を用いた．

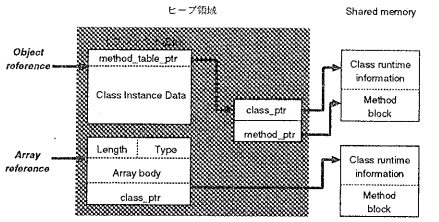

図 16 ハンドルを除去したメモリ構造

6.1 Allocation History GC

Allocation History GC をインプリメントした JVM を用いて測定を行った．アロケーション履歴バッファは 8192 エントリ，参照ビットテーブルは 1 オブジェクトにつき 2 ビット用意した．

Allocation History GC によるオブジェクト回収率を次式で定義する．

$$\frac{\text{Allocation History GC で回収したオブジェクト数}}{\text{アロケートしたオブジェクト総数}}$$

javac の場合 Allocation History GC によるオブジェクト回収率は 41.5%となった．参照ビットデクリメント時に深さ 1 の走査しか行っていないにも関わらず，高い回収率が得られている．

全実行時間，Mark & Compaction (表では M & C) の起動回数，Mark & Compaction 1 回当りの処理時間，Allocation History GC (表では A. H. GC) の起動回数，Allocation History GC 平均処理時間，を表 1 に示す．Mark& Compaction GC の起動回数が 8.1% 削減されていることがわかる．Allocation History GC に要した時間は全実行を通して 1.1 秒であるから，GC 全体の処理時間は，3.1% 削減されている．全実行時間をみると 2.5% 短縮している．比較的回収率 (41.5%) がよいにもかかわらず，Mark& Compaction の起動回数が 19%しか減少していない．これは Allocation History GC を行なっても依然としてヒープフラグメンテーションの問題が生じてしまうためであると考えられる．

表 1 Allocation History GC の効果

	A. H. GC	オリジナル JVM
全実行時間	85.9 秒	86.9 秒
M&C 処理時間	33.4 秒	35.6 秒
M&C 起動回数	102 回	111 回
平均 M&C 処理時間	327.3 ms	320.4 ms
A. H. GC 処理時間	1.1 秒	– ms
A. H. GC 起動回数	279 回	– 回
平均 A. H. GC 処理時間	3.9 ms	– ms

6.2 Hasty Compaction

この測定ではフラグメンテーション度の閾値を 100

としてある．メモリ使用量を図 17 に示す．総空きサ

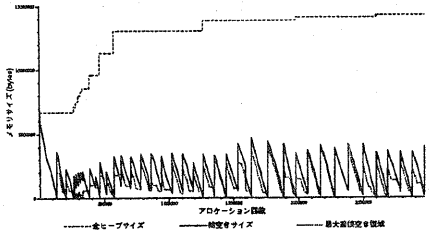

図 17 Hasty Compaction を用いた場合のメモリ使用量

イズが増加している箇所が GC を行った箇所である．GC の起動回数が図 7 に対して大幅に減っていることがわかる．

全実行時間，GC の起動回数，GC1 回当りの処理時間を表 2 に示す．全実行時間が 28.0% 短縮している．

表 2 Hasty Compaction の効果

	Hasty Compaction	オリジナル JVM
全実行時間	62.6 秒	86.9 秒
GC 処理時間	25.1 秒	35.6 秒
GC 起動回数	34 回	111 回
平均 GC 処理時間	739.4 ms	320.4 ms

実行時間の短縮の要因として GC 処理時間の短縮が挙げられる．平均 GC 処理時間は 2.3 倍要するが，延起動回数が 30%程度に低下しているため，全 GC 処理時間は，29.5% 短縮している．

実行高速化の別の要因としてアロケーションの高速化が考えられる．オブジェクトのアロケート時は，next fit アルゴリズムによって空き領域を見つける．このため連続した空き領域サイズが大きいほど，空き領域のサーチが速くなる．Hasty Compaction は比較的速く連続した空き領域を作るため，アロケーションが速くなる．

6.3 Segregated Memory Management

Segregated Memory Management をインプリメントした JVM で性能測定を行なった．全実行時間，GC の起動回数，GC1 回当りの処理時間を表 3 に示す．

表 3 Segregated Memory Management の効果

	Segregated MM	SUN JVM
全実行時間	45.1 秒	86.9 秒
GC 処理時間	11.6 秒	35.6 秒
GC 起動回数	58 回	111 回
平均 GC 処理時間	201.ms	320.4 ms

この測定結果から実行時間は 48.0% の削減，GC 処理時間は 67.2%の削減に成功していることがわかる．実行時間の削減の要因には主に次の 3 つが考えられる．() 内には短縮した時間を示してある．

- GC 処理時間の短縮 (-24 秒)
- ハンドル参照時間の除去 (-5.1 秒)
- アロケーション時間の短縮 (-6.1 秒)

Segregated Memory Management によってアロケーション時間も短縮できている．javac のような複雑なアプリケーションでは，メモリフラグメンテーションが生じやすい．このため本方式のようなフリーリストによる空き領域管理の効果があらわれたものと思われる．

1 回の GC の実行時間内訳を図 18 に示す．ブロッ

図 18 GC の実行時間内訳

クのマージに要する時間は非常に短いことがわかる．(平均で 1ms) また Valid ビットマップを用いたことによって Mark フェーズと Sweep フェーズが短縮されている．

ヒープ使用量を図 19 に示す．ヒープの増大が 28.4% 程度で押さえられていることがわかる．

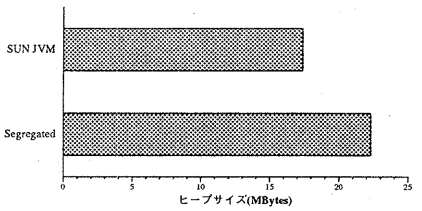

図 19 GC のヒープ使用量

6.4 SUN の JIT コンパイラとの比較

本稿では JVM の性能比較を行なうために，我々が開発した JIT コンパイラを使用してきた．ここでは SUN の JIT コンパイラ (**sunwjit**) を使用したシステムと比較し，他システムとの相対的な比較を行なう．我々の JVM として Segregated Memory Management を使用したものを使用している．

javac を実行したときの実行時間を表 4 に示す．SUN の JIT コンパイラを使用した場合と比較して，実行速度は 2.3 倍となった．

表 4 SUN の JIT との比較 (javac)

	Segregated MM	sunwjit
全実行時間	45.1 秒	101.7 秒
GC 処理時間	11.6 秒	35.5 秒
GC 起動回数	58 回	122 回
平均 GC 処理時間	201.0 ms	326.6 ms

6.5 考　察

ここまで 3 つの方式を提案してきた．オリジナルの JVM との相対的な値を表 5 にまとめる．これら 3 手

表 5 SUN の JVM との比較

	全実行時間	GC 処理時間	平均 GC 処理時間	ヒープサイズ
AHGC	98.8%	96.9%	28.2%	100.0%
HC	72.0%	70.5%	161.6%	90.5%
SMM	51.9%	32.6%	62.7%	128.4%

AHGC: Allocation History GC
HC: Hasty Compaction
SMM: Segregated Memory Management

法のなかで Segregated Memory Management が最も高速であることがわかる．Allocation History GC は GC 起動回数が多いために，GC1 回当りの処理時間は短いが，javac のようなメモリフラグメンテーションが生じやすいアプリケーションでは効果が少ない．

Segregated Memory Management の欠点はより大きなヒープメモリを消費することにある．一般にヒープメモリが大きいと GC の起動回数が少なくなるため，このままの比較では方式を比較したことにはならない．そこでオリジナルの JVM の初期ヒープサイズを変更して実行時間を測定した．(ベンチマークは javac) 図 20 に結果を示す．64Mbytes から 128Mbytes にかけ

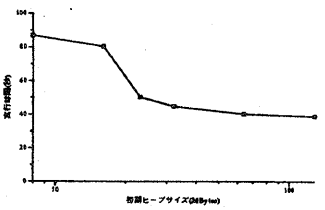

図 20 初期ヒープサイズによる実行時間の変化

て実行時間の減少がほとんどみられない．javac ベンチマークでは初期ヒープサイズが 64Mbytes で GC 起動回数が 1 回，128Mbytes で GC 起動回数が 0 回になる．したがってこれ以上実行時間は短縮できない．Segregated Memory Management と同等の実行速度を得るためには 32Mbytes の初期ヒープサイズが必要であることがわかる．したがって本メモリ管理方式は，オリジナル JVM と比較して明らかに高速であると言える．

7. ま と め

今回我々は Java VM の GC の高速化のために，3 つのガベージコレクション・メモリ管理方式 (Allocation History GC, Hasty Compaction, Segregated Memory Management) を提案した．またこれらの方式を JDK1.2beta3 をベースにインプリメントし性能比較を行なった．

提案した 3 方式の中でとくに Segregated Memory Management が有効であった．このメモリ管理は，javac をベンチマークテストとした場合，実行時間を 48%短縮し，GC 処理時間を 67.2%短縮する．

今後は Segregated Memory Management を使用した Java VM をベースに，GC の高速化・リアルタイム化 (GC1 回当りの処理時間の短縮) を行なっていく予定である．

参 考 文 献

1) 志村他, Java の高速化, 第 39 回プログラミング・シンポジウム報告集, p99-108, 1998
2) C.-H.A. Hsieh, M.T. Conte, T.L. Johnson, J.C. Gyllenhaal, W.W. Hwu, Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results, Proc. of the 29th Annual International Sympo. on Microarchitecture, p1-8, 1996
3) A. Krall and R. Grafl, CACAO - A 64 bit JavaVM Just-in-Time Compiler, PPoPP'97 Workshop on Java for Science and Engineering Computation, Available at http://www.complang.tuwien.ac.at/java/cacao/
4) M.S. Johnstone , P.R. Wilson, The Memory Fragmentation Problem: Solved?, Proc. of OOPSRA 1997 Workshop on Garbage Collection and Memory Management, Available at http://www.dcs.gla.ac.uk/ huw/oopsla97/gc/papers.html
5) Paul R. Wilson, Uniprocessor Gabage Collection Techniques, Available at ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps
6) Sun Microsystems, Inc: The Java Virtual Machine Specification