A-010

明示的なメモリ管理機能を備えたJava仮想マシンの評価

Evaluation of Java Virtual Machine with Explicitly Managed Heap Memory

足立 昌彦† 小幡 元樹† 西山 博泰†
Masahiko Adachi Motoki Obata Hiroyasu Nishiyama
岡田 浩一‡ 中島 恵‡
Koichi Okada Kei Nakajima

1. はじめに

Java[1] は高い移植性や再利用性などの特長により，組み込みシステムからサーバシステムに至るまで，様々な領域への適用が進んでいる．特にサーバシステムでは，Web システムだけでなくミッションクリティカル・システムへの応用も進んでいる．このようなシステムでは，近年，大容量メモリを搭載することが多く，今後更なる大容量化の進展が予想される．

Java仮想マシン(Java Virtual Machine: JVM)では，ガベージコレクション(GC)により不要なメモリ領域を回収する．ミッションクリティカル・システムへ Java を適用する場合，高い信頼性と性能が要求される．そのためスループット性能・安定性の観点から，stop-the-world 型の世代別 GC[2] を採用することが多い．応答性を要求するサーバでは，Major GC によるレスポンス性能の低下が問題となる．特に大容量メモリを有する場合は，この問題が顕著となる．

そこで我々は Major GC の発生頻度を低減する方式の検討を行っている．世代別 GC では，短寿命オブジェクトは New 領域内に留め，不要となった時点で Minor GC により回収し，長寿命，特にシステム停止まで利用し続けるようなオブジェクトは Tenured 領域に昇格させ，New 領域を圧迫しないのが理想である．ここで問題となるのが，明確に寿命が決まっているが，Tenured 領域に昇格する程度に生存期間が長い中寿命オブジェクトである．このようなオブジェクトは，Tenured 領域を不要に圧迫し，Major GC の発生頻度を高める．

このような中寿命オブジェクトに着目し，我々はソースコードレベルでメモリ管理を可能としたヒープ領域(Explicit Heap: Eheap)を有する JVM(JVM-EH)を提案している．JVM-EH は，Eheap 内メモリ領域(Explicit Memory: EM)の確保，解放および EM 内へのオブジェクトの配置制御を Java プログラムから可能とする API と，それに対応したメモリ管理を実現する．Eheap には複数の EM を確保することができ，さらに，EM 内に複数のオブジェクトを配置することができる．EM を解放することで，EM 内に配置された複数のオブジェクトを一括削除することを可能とする．この機能を利用し，中寿命なオブジェクトを EM に配置し，それらが不要となった時点で EM を削除する．これにより Tenured 領域へのオブジェクト蓄積を抑制し，Major GC の発生回数を削減可能とする．本稿では，我々が提案する明示的なメモリ管理機能を備えた JVM の評価を行い，その効果と特性を示す．

†（株）日立製作所 システム開発研究所
‡（株）日立製作所 ソフトウェア事業部

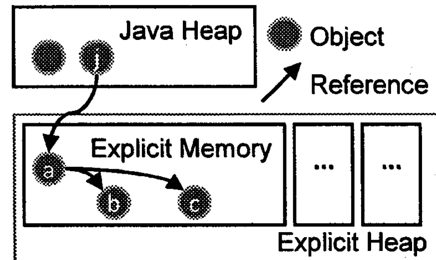


図 1 JVM-EH におけるメモリ領域の構成

2. 明示的なメモリ管理機能を備えた Java VM

我々が提案する JVM-EH は，API を通してメモリ領域の確保，解放およびオブジェクトの配置を制御できる．

JVM-EH が管理するメモリ領域の構成を図 1 に示す．

JVM-EH は Java Heap(Jheap)と Explicit Heap(Eheap)の 2 つのメモリ領域を持つ．Jheap は従来の JVM が管理するヒープ領域である．Eheap は JVM-EH が新たに提供する API を通してメモリ管理可能なヒープ領域である．Eheap の主な利用例は，まず Eheap から EM を確保し，そこへオブジェクトを複数配置する．配置したオブジェクトが不要になった時点で EM を解放する．EM の解放は，そこに存在するオブジェクトを一括削除することを意味する．このように，メモリ領域管理の一部を明示的にプログラマが行えるため，メモリ利用効率を高めることができる．

2.1 Explicit Memory の確保と解放

EM を確保するには JVM でオブジェクトを生成する場合と同様に new により EM オブジェクトを生成する．JVM-EH は EM オブジェクトの生成と同時に，Eheap から EM を確保する．なお EM へ配置するオブジェクトのサイズが確保した領域に収まらない場合，JVM-EH はそのオブジェクトが配置できるよう EM を拡張する．

EM の解放は API を利用してソースコード上で明示的に行う．これにより，解放領域に含まれるオブジェクトを一括削除できる．ただし，大規模なシステムでは，オブジェクトの全ての参照関係を把握することは難しい．また，フレームワーク等を利用する場合，その内部においてプログラマが意図しない参照が作成されることがある．このことから，EM 解放時に解放対象メモリ上のオブジェ

[1] Java は，米国 Sun Microsystems, Inc.の米国およびその他の国における商標または登録商標です．

[2] 世代別 GC では Java ヒープを New 領域と Tenured 領域に分割して管理し，New 領域の使用量が閾値を超えたとき，Minor GC により New 領域の回収を行う．Minor GC を数回繰り返した後でも生存しているオブジェクトは長寿命オブジェクトとみなし，Tenured 領域に移動する．これを昇格という．Tenured 領域の使用量が閾値を超えると Major GC により，Java ヒープ全体の領域回収を行う．

クトへの領域外からの参照が存在する場合が考えられる．この状態で EM を単純に解放すると dangling pointer が発生する可能性がある．すなわち，解放済オブジェクトへの不正参照が生じる．例えば図 1 のような参照関係が残った状態で EM を解放すると，オブジェクト j から a への参照が不正になる．

そこで，JVM-EH では EM 解放時に解放対象内のオブジェクトへの領域外からの参照の有無を検査し，被参照オブジェクトが存在する場合は，それを解放対象外メモリへ移動する．解放対象メモリ上のオブジェクトへの領域外からの参照がなくなるまでこの処理を繰り返した後，EM を解放する．これにより，安全なメモリ解放を実現する．例えば，図 1 の j のようなオブジェクトが存在すると，そこから参照できる全てのオブジェクト(a，b，c)を解放対象外メモリ(例えば Jheap)へ移動することで，EM を安全に解放することができる．

2.2 オブジェクトの配置方法

JVM-EH における EM へのオブジェクト配置方法は以下の 2 つである．

Context 型

オブジェクトの配置を開始するメソッドと終了するメソッドの間で生成したオブジェクトを全て EM への配置対象とする．一般に，ライブラリを利用したプログラムでは，その内部で生成する一時オブジェクトをプログラマが把握することは困難である．そのため，メソッド間で生成した全てのオブジェクトを EM へ移動すると，一時オブジェクトも移動対象となる．これは不要な移動オーバヘッドとなる．これを回避するため，移動の契機を Minor GC 実行後とし，EM への一時オブジェクトの不要な移動を抑止している．この配置方法は，生成するオブジェクト間に参照関係が無く，また，生成箇所がソースコード上で集中している場合に適している．

```
1: ContextExplicitMemory cem =
            new ContextExplicitMemory();
2: cem.enter();
3: o1 = new Obj(); o2 = new Obj();
   o3 = new Obj(); o2 = null;
4: cem.exit();
5: { any operation; }
6: cem.reclaim();
```

図 2 Context 型の使用例

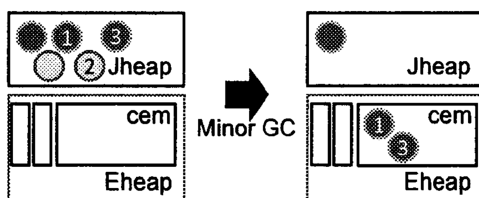


図 3 Context 型のオブジェクトの配置

ソースコード例と，それに対応するメモリ状態をそれぞれ図 2，3 に示す．1 行目のコンストラクタで Eheap 内に EM(cem)を確保する．2 行目の enter()から 4 行目の exit()の間で生成されるオブジェクト(1,2,3)が cem への移動対象となる．オブジェクト 2 を Minor GC の回収対象とする．5 行目を実行中に Minor GC が発生した場合，回収対象とならない移動対象オブジェクト 1,3 を cem に移動する．6 行目の reclaim()で，cem とそれに含まれるオブジェクト 1,3 を解放する．

Reference 型

あるオブジェクトから参照できるオブジェクトの生存期間は同程度であることが多い．そこで，Reference 型の配置方法では，先頭となるオブジェクトを EM に配置し，その先頭オブジェクトから参照を辿れる全てのオブジェクトを自動的に EM への配置対象とする．ただし，このオブジェクト集合が短寿命であった場合，それらを EM に移動すると，移動時間および EM の空間が無駄となる．そこで，移動の契機をオブジェクトが昇格する時とし，EM への短寿命オブジェクトの移動を抑止している．この配置方法は，生成オブジェクト間に参照関係があれば，生成箇所がソースコード中で分散している場合でも適用しやすい．

```
1: RerenceExplicitMemory rem =
            new ReferenceExplicitMemory();
2: ArrayList al = rem.newInstrance(ArrayList.class);
3: Obj o1 = new Obj(); al.add(o1);
   Obj o2 = new Obj(); al.add(o2);
4: rem.reclaim();
```

図 4 Reference 型の使用例

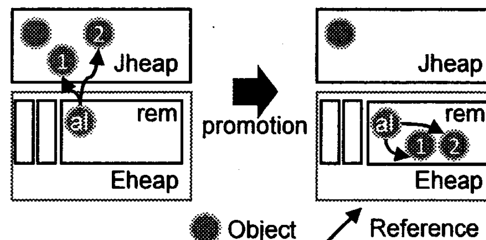


図 5 Reference 型のオブジェクト配置

ソースコード例と，それに対応するメモリ状態を図 4,5 に示す．1 行目のコンストラクタで，Eheap から EM(rem)を確保する．2 行目の newInstance()で先頭オブジェクト(al)を生成し rem に配置する．al はこの時点で rem に配置される．3 行目で，al から参照できるオブジェクト(1,2)を生成するが，この時点では Jheap に配置される．4 行目の reclaim()で rem を解放するまでに al から参照できるオブジェクトが昇格する場合，それらを rem に移動する．

3. 評価

3.1 評価環境

本稿では以下に示す 2 つの評価を行う．JVM と JVM-EH とを比較する． 評価環境は，OS/CentOS4.4，CPU/Xeon 2.4GHz*2，Memory/2GB である．評価で使用する合計ヒープサイズはいずれも 1024MB とする．JVM では 1024MB を Jheap に割り当てる．JVM-EH では 896MB を Jheap に，残りの 128MB を Eheap に割り当てる．

(A)基礎評価

(A)は提案手法の基礎的な評価を行うことを目的としている．特に，EM の確保，オブジェクトの移動が発生する Minor GC の性能および EM 解放において，その性能を評価する。Minor GC 性能の評価では，移動するオブジェクト量を{0.25MB, 0.5MB, 1MB, 2MB, 4MB}と変化させて評価する．なお，移動するオブジェクトは同一サイズのオブジェクトをノードとした線形リストである．Minor GC が発生すると，JVM ではリストは Tenured 領域に移動し，JVM-EH では EM に移動する．EM 解放における性能評価では，EM に 4MB のリストが存在する状態を想定し，その EM を解放する際の解放対象外メモリへの移動量を{0.25MB, 0.5MB, 1MB, 2MB, 4MB}と変化させて評価する．

(B)Tomcat

(B)は，実際のサーバ・クライアント・システムにおいて提案手法を評価する．アプリケーションサーバの Tomcat(Tomcat5.5.12)上でアプリケーションを動作させる．この Tomcat 上のアプリケーションはログインしたユーザ毎に XML 形式の文字列をパースし DOM(Document Object Model)を構築する．ユーザがログアウトすると，その DOM オブジェクトは不要になるとする．ここではユーザのログインからログアウトまでをシナリオと定義する．シナリオ毎に EM を確保し，オブジェクトを配置するようにプログラムを修正した．Tomcat はログインしたユーザ毎にデータを保持するオブジェクトを生成する．これを先頭オブジェクトとして Reference 型のメモリ配置を適用する．Web アプリケーションには様々な種類があり，シナリオ毎に生成するオブジェクト量も異なっている．よって生成する DOM の要素を{1000, 5000, 10000}と変化させて実験を行う．また，ユーザのアクセス時間が均一であることは考えにくいため，短時間でシナリオが終わるユーザと長時間でシナリオが終わるユーザを想定する．短時間のシナリオは，ユーザがログインして，DOM の構築が終わるとすぐにログアウトする．つまりサーバの処理性能が高いほど，多くのシナリオを処理できる．これにより，スループット性能を評価する．なお，本実験において，短時間シナリオは 1 秒未満であった．長時間シナリオはユーザのログインからログアウトまでの時間を 15 秒とする．短時間シナリオと長時間シナリオのユーザ比率を 97 : 3 とし，JVM-EH の効果を評価する．

3.2 結果と考察

評価(A)の結果を図 6 に示す．図 6-(a)は JVM と JVM-EH に関して，移動するオブジェクト量に対する Minor GC の処理時間を示したグラフである．JVM-EH では JVM に比べ EM へオブジェクトを移動するために一定のオーバヘッドを要していることが分かる．このオーバヘッドは移動するオブジェクト量に依存せず，Minor GC はオブジェクトの移動量に比例して処理時間が大きくなっている．そのため，移動するオブジェクト量が大きくなるほど，JVM-EH のオーバヘッドは相対的に小さくる．オブジェクト移動量が 4MB の場合，JVM-EH は JVM に比べて，Minor GC 時間が約 4.6%増大している．

図 6-(b)は JVM-EH において，EM 解放処理で解放対象外領域へのオブジェクト移動量に対する処理時間を示したグラフである．図 6-(a)(b)を比較すると，安全な EM 解放の処理時間は Minor GC の約 1/10 である．解放の際，解放対象外領域へ移動するオブジェクト量に比例して，処理時間は増大している．また，1 つの EM を確保する処理時間は 1 ミリ秒未満であった．

評価(B)の結果を図 7 に示す．図 7-(a)は JVM と JVM-EH における Tenured 領域の使用量の推移である．JVM に比べ JVM-EH の Tenured 領域の使用量は低くなっている．シナリオで生成するオブジェクト量の変化に対しても，Tenured 領域の使用量の増加量を安定して低く抑えることができている．これは，シナリオ毎に生成するオブジェクトがシナリオ終了時に全て不要となり，解放対象外メモリへの移動が発生しないためである．これにより，JVM-EH は Major GC 発生回数を削減できている．

JVM-EH における Eheap 使用量の推移を図 7-(b)に示す．シナリオ毎に生成するオブジェクト量が異なっていても，EM の使用量は全て安定している．これは，シナリオ終了後に，そのシナリオで生成した EM を解放することで，Eheap を効率的に利用できるためである．

評価(B)でシナリオ毎に生成する要素数が 10000 の場合の性能評価結果を表 1 に示す．

JVM では 1 回に 0.908 秒停止する Major GC が 7 回発生しているのに対し，JVM-EH では発生を抑止できている．また，JVM-EH における Minor GC の停止時間は JVM 比べ 2%程度増加している．

EM 解放の 1 回あたりの停止時間は Major GC の 1/50 程度であり，Minor GC の 1/10 程度に抑えられている．これは図 7-(A)の結果で示したとおり，EM 解放時に解放対象外メモリ上のオブジェクトから解放対象メモリ上のオブジェクトへの参照が殆ど存在しないためである．

表 1 Tomcat 上で動くプログラムの評価結果

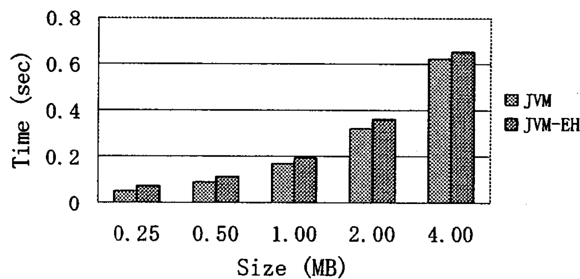
	JVM		JVM-EH	
	秒/回	回数	秒/回	回数
Major GC	0.908	7	N/A	0
Minor GC	0.191	628	0.195	599
EM 解放	–	–	0.021	589
スループット		117064		104484

JVM-EH のスループット性能は JVM に比べ，約 1%の低下となっている．また，評価(A)と同様に，1 つの EM を確保する処理時間は 1 ミリ秒未満であった．

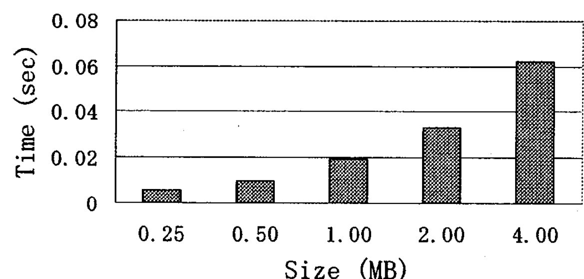
JVM-EH では Minor GC の処理時間の増加や，EM 解放にごく短い停止時間を要するが，Major GC の発生回数を 0 回にすることができており，スループット性能を約 1%の低下に抑えつつ，レスポンス性能を約 5 倍にすることができた．ここではシナリオ毎に生成する要素数が 10000 の結果のみを示しているが，その他の場合においても同様の傾向を得ている．

4. 関連研究

C/C++といった言語では malloc/free などのメモリ管理 API を利用して明示的にメモリの動的確保/開放を行い，その安全性はプログラマが保証する必要がある．メモリ管理性能の向上や管理容易化の研究が様々になされている[1]が，その 1 つとして，リージョンベース手法が提案されている．リージョンベース手法では，我々の提案手

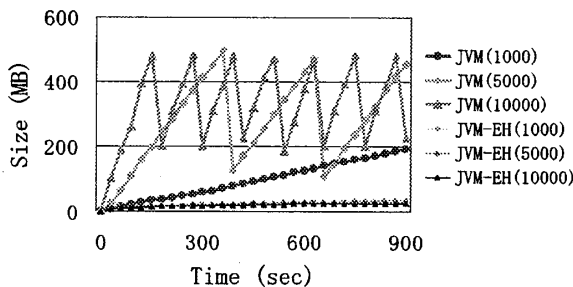


(a)Minor GC時間

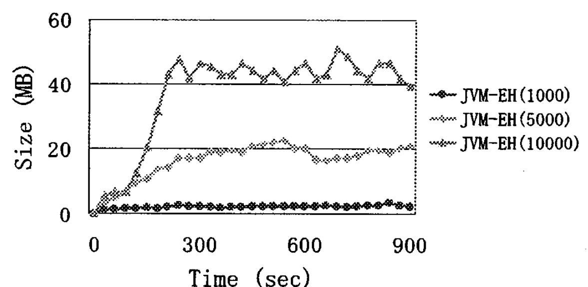


(b)EM解放時間

図6 基本性能評価結果



(a)Jheapの使用量



(b)Eheapの使用量

図7 Tomcatの実験結果

法と同様に領域の一括削除が可能であるが，安全性をプログラマが保証する点に変わりはない．

Region-Based Memory Management[2]では関数型言語の一種であるMLのサブセットを対象とし，オブジェクトをスタック的に管理するリージョンに割り付けることでGCを利用しないメモリ管理手法を提案している．この手法では，型推論に基づいたプログラム解析により，各オブジェクトを配置すべきリージョンを求める．Javaのような手続き的かつ複雑な言語仕様を持つシステムへ，この手法をそのまま適用することは困難であると考えられる．

Real-Time Specification for Java(RTSJ)[3]はリアルタイム処理を考慮したJavaの拡張仕様である．RTSJではScoped Memoryと呼ばれるメモリ領域を用いることで，スタック型のメモリ管理を可能としている．これによりJheapへのオブジェクト生成を削減することでGC発生の制御を可能としている．Scoped Memoryの解放に伴うdangling pointerの発生を防ぐため，RTSJではオブジェクト間の参照関係に関する制約を設けている．すなわち，階層的な関係を持つScoped Memory内に確保されたオブジェクトに関し，外側のScoped Memory内のオブジェクトが内側のScoped Memory内のオブジェクトへの参照を保持することを実行時に検知し例外を発生させる．よって，従来のJavaプログラムに対しScoped Memoryを用いたプログラムは互換性やスループット性能が低下するという問題がある．

権藤らはリアルタイム処理に適したGC機能として，GC APIを提案している[4]．このAPIを利用することで，リアルタイム制約を満たすために，指定区間内でGCの発生を抑止することができる．Context型と同様に，外部ヒープからメモリ領域を確保し，区間内で生成したオブジェクトをその外部メモリに配置する．これにより，Javaヒープの圧迫を抑止でき，その区間においてGCを発生させなくできる．外部メモリ使用後はAPIにより解放するが，その際の安全性の保証はない．そのため，解放対象領域への領域外からの参照が存在しないことをプログラマが保証する必要がある．また，Context型とは異なり，一時オブジェクトを含め，指定区間で生成された全てのオブジェクトを外部メモリに配置する．これは外部メモリの利用効率の低下を招くことが予想される．

5. おわりに

本稿では，我々が提案している明示的なメモリ管理機能を有するJava仮想マシンについて評価を行った．提案手法で提供するAPIを利用し，メモリ管理を行うよう修正したプログラムでは，オブジェクトの移動と解放のオーバヘッドを要するが，スループット性能を約1%の低下に抑えることができ，かつ，従来手法に比べてMajor GCの発生回数を削減することができた．

本稿では，実アプリケーションにおける評価を行っていない．今後，実際のアプリケーションを用いて，提案手法の有効性を検証していく必要があると考える．

参考文献

[1] Emery D. Berger, Benjamin G. Zorn and Kathryn S. McKinley. Reconsidering custom memory allocation. In Proceedings of the 2002 Conference on Object-Oriented Programming: Systems, Languages and Applications, 2002.

[2] Mads Tofte and Jean-pierre Talpin. Region-Based Memory Management. Information and Computation, 1997.

[3] Filip Pizlo, J. M. Fox, David Holmes and Jan Vitek, Real-Time Java Scoped Memory: Design Patterns and Semantics, Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2004.

[4] 権藤勝彦, 八十島利典, 喜多山卓郎. リアルタイム環境に適したガベージコレクションAPIの実現について, 第4回プログラミングおよび応用のシステムに関するワークショップ(SPA2000), 2000.