情報処理学会論文誌　プログラミング　Vol. 3　No. 2　26–35 (Mar. 2010)

Javaにおける明示的メモリ管理領域の半自動適用技術

足立昌彦[†1]　小幡元樹[†1]　西山博泰[†1]
岡田浩一[†2]　長瀬卓真[†2]　中島恵[†2]

Javaは GC による自動メモリ管理を採用しているが，ミッションクリティカルな分野に適用する場合，長時間 GC によるアプリケーションの停止が問題となる．これに対して，我々は明示的なメモリ管理を Java に導入することにより，長時間 GC によるアプリケーション停止回数を削減する手法を提案している．本手法は，API を通してメモリ領域の生成，削除，およびメモリ領域へのデータ配置制御をユーザが行う．本論文では，ソースコードを修正せずに明示的メモリ管理を適用するための技術を提案する．明示的メモリ領域の生成については，外部からその生成点を指定することにより適用し，解放についてはメモリの利用状況を Java 仮想マシンが監視し，不要と判断した時点で自動的に解放する．提案手法を大規模なオープンソースのミドルウェアなどに適用した結果，ソースコードを修正することなく実用的な実行性能を維持したまま，明示的メモリ管理を適用でき，長時間 GC の発生頻度を抑止できることを確認した．

Semi-automatic Application of Explicitly Managed Heap Memory for Java

Masahiko Adachi,[†1] Motoki Obata,[†1]
Hiroyasu Nishiyama,[†1] Koichi Okada,[†2]
Takuma Nagase[†2] and Kei Nakajima[†2]

This paper proposes the semi-automatic application of explicitly managed heap memory for Java. The explicitly managed heap memory is a kind of heap memory aimed at reducing long waiting time caused by stop-the-world style garbage collection. The proposed heap memory can be managed by several APIs from Java programs. The APIs are designed to be used without sacrificing execution performance and ease of programming. We propose techniques that allow application of the explicitly managed heap memory without source code modification. Allocation of the explicitly managed heap memory is controlled by specifying target points externally. Deallocation of the heap memory is performed automatically according to its necessity. By applying these techniques to several applications, we observed that long pause time can be controlled without source code modification.

1. はじめに

近年，Java[1),⋆1]システムがサーバ分野における業務プログラムの基盤や，電子商取引に代表されるミッションクリティカル性の高い基幹システムなどで使われるようになってきている．これらの領域ではプログラムを実行する Java 仮想マシン（Java VM）に対し，高い信頼性と実行性能が求められる．このため，アプリケーションの実行性能が高く利用実績の豊富な stop-the-world 型の世代別ガベージコレクション（以下，GC）を採用することが多い．GC が発生するタイミングは，プログラム中で生成されるデータを格納するヒープメモリの使用量が閾値を超える直前であるが，このタイミングを予見することは容易でない．また，世代別 GC では処理時間の短い Minor GC と長時間の停止を要する Major GC がある．これらの停止時間は利用ヒープメモリ容量に応じて長くなる傾向にある．近年のハードウェアリソースの増大により，Java が利用できるヒープメモリ容量は増加しており，GC による停止時間の長期化は顕著になっている．特に Major GC によって生じる長時間停止はミッションクリティカル性を要求するシステムにおいては許容し難いケースが多々ある．この問題を解決するため，我々は Java プログラム上からメモリ領域の確保・解放を明示的に指示可能な明示的メモリ管理 API を提案している[2)]．これは，データ群を領域ごとに管理する API をプログラマが明示的に利用することにより，Major GC の発生頻度を抑止するものである．

一方，近年の Java システム開発は，大規模なライブラリやフレームワークを組み合わせて開発・構築することが多い．従来の明示的メモリ管理は，我々の提案する API をサポートする Java VM でのみ利用できる．そのため，提案 API を適用した場合，他の Java VM と

†1 日立製作所システム開発研究所
System Development Laboratory, Hitachi Ltd.
†2 日立製作所ソフトウェア事業部
Software Division, Hitachi Ltd.
⋆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．

の可搬性が低下するという問題がある．また，すでに完成しているシステムにおいて，そのプログラムを修正し再構築することなく明示的メモリ管理を適用したいという要求もある．

本論文では，明示的メモリ管理 API をソースコード修正することなく適用する方法を提案する．また，本手法をベンチマークプログラムや大規模なオープンソースのミドルウェアプログラムに適用した結果について述べる．

2. 明示的メモリ管理

本章では，明示的メモリ管理 API について述べる．なお，以後の説明で述べる Java VM は HotSpot VM[3] を対象としている．

2.1 明示的メモリ管理の概要

世代別 GC を採用した Java VM は，ヒープメモリを新世代領域（New 領域）と旧世代領域（Tenured 領域）に分割して管理する．HotSpot VM では，New 領域不足時に実行する GC（Minor GC）として Copy GC 手法を採用し，Tenured 領域不足時に実行する GC（Major GC）として Mark-Compact GC 手法を用いている．いずれの GC 手法も Java プログラムを停止して実行する stop-the-world 型である．一般に，Major GC の処理時間は Minor GC の処理時間の十数倍から数十倍であり，システムからの応答が長時間なくなってしまうという問題がある[⋆1]．世代別 GC において Minor GC を複数回実施しても回収できなかったオブジェクトは Tenured 領域へ移動（promotion）する．この promotion により Tenured 領域の使用量が増加する．Tenured 領域の使用量が閾値を超えると Major GC が発生する．そのため，Tenured 領域へのオブジェクト移動頻度を抑えることができれば，Major GC の発生回数を削減することが期待できる．このような観点から，我々は明示的メモリ管理を提案している．明示的メモリ管理とは，Java プログラムで利用するオブジェクト群を GC 対象外メモリ領域（Explicit メモリ）に配置し管理する仕組みである．

ユーザは明示的メモリ管理 API を通して Explicit メモリの生成，削除およびそのメモリへのオブジェクト配置などを行う．Explicit メモリは管理用 Java オブジェクトと関連付けられており，管理オブジェクトの生成により暗黙的に領域を確保する．Explicit メモリへのオブジェクト配置の際には，領域の自動拡張を行う．Explicit メモリへのオブジェクト配置方法としては Context 型と Reference 型の 2 種類を提供する．

Context 型 Context 型は処理に着目した配置方法である．すなわち，enter/exit というメソッドで挟んだ区間の実行過程で生成されたオブジェクトを Explicit メモリへ配置する．配置したオブジェクト群が不要になった時点で解放指示（reclaim）を出し，その領域ごと解放する．

Reference 型 Reference 型はデータ構造に着目した配置方法である．すなわち，データ構造の基点となるオブジェクトを指定し，そこから参照可能なオブジェクトを Explicit メモリへ配置する．Context 型と同様に，配置したオブジェクト群が不要になった時点で解放指示を出し，領域ごとオブジェクトを解放する．

Explicit メモリへのオブジェクト配置は Minor GC における promotion 時に実施する．すなわち，従来の promotion によるオブジェクトの移動先が Tenured 領域から Explicit メモリ領域に変更になるだけで，処理時間に大きな影響はなく Minor GC によるプログラム停止時間の劣化は少ない．Explicit メモリの解放は Java プログラムを停止し，その領域に配置された複数のオブジェクトを一括して削除する．ただし，生存（live）オブジェクトを削除するとプログラムの動作が不正になる可能性がある．このため，live である可能性のあるオブジェクトを Java VM が検出し，解放対象外のメモリ領域へ移動する[⋆2]．以上で説明した明示的メモリ管理 API を用いることにより，これまで Tenured 領域に配置されていたオブジェクトを Explicit メモリに配置することが可能になる．これにより，Major GC による長時間停止を抑止できる．

2.2 明示的メモリ管理の課題

従来の明示的メモリ管理には次の 2 つの課題がある．

(1) **ソースコードの修正**

1 つ目の課題は，ソースコードを修正できない場合に，明示的メモリ管理を利用できないことである．前述のとおり，明示的メモリ管理は API を通して利用する．そのため，利用者はソースコードを修正する必要があるが，ライセンスや運用上の制約などから，そのソースコードを修正できない場合がある．この課題に対し，明示的メモリ管理の適用を完全に自動化することが考えられる．すなわち Explicit メモリの適用対象とする処理やデータ構造を実行時に解析し，対象データを自動的に Explicit メモリへ配置する．また，Explicit メモリ

⋆1 HotSpot VM では，Concurrent Mark-Sweep（CMS）GC をもサポートしている．CMS はミューテータの実行と並行して GC 処理を行うため停止時間の削減が可能であるが，実行性能が低下することや，メモリ領域の断片化が進むと，その解消のために長時間の停止を必要とするため，本論文で対象とするシステムでの利用には適していないと思われる．

⋆2 live である可能性のあるオブジェクトが多く存在する Explicit メモリを解放する場合，多数のオブジェクト移動により，処理時間が長くなる．これは Java プログラムの停止時間を長期化させる要因になるため，注意が必要である．

に配置されたオブジェクトが不要になった時点で，その領域を解放する．これにより，ソースコードを修正せずに明示的メモリ管理を適用できる．しかし，このような解析は容易ではなく，可能な場合でも解析に長い時間を要することが容易に想像できる．そこで，Explicit メモリの適用箇所を外部から指定し，指定箇所を含むバイトコードを Java VM が読み込むとき，Explicit メモリを利用するようバイトコードを書き換える方法を提案する．この方式では，バイトコードを書き換えるための処理時間を要する．しかし，その処理はバイトコード読み込み時のみであるため，実行時性能への影響は比較的小さく抑えることができる．

(2)　**Explicit メモリの効果的な解放**

もう 1 つの課題は，Explicit メモリの効果的な解放点の指定が容易でないことである．前述のとおり，Explicit メモリの解放は，領域に live オブジェクトが多数存在する場合，解放処理に多くの時間を要する．そのため，効率的な解放には Explicit メモリに配置されたオブジェクトの多くが不要になるタイミングを把握する必要がある．しかし，フレームワークなどを用いた大規模なプログラムでは，オブジェクトの要否を把握することが容易でない場合がある．この課題に対し，我々は，Explicit メモリの自動解放方法を提案する．すなわち，Java VM が Explicit メモリを監視し，その内部に配置されたオブジェクトの多くが不要（dead）であると判断したときに Explicit メモリを自動で解放する．

3. 明示的メモリ管理の半自動適用方法

本章では，Explicit メモリの生成点を外部から指定するための方法と，Explicit メモリ領域の解放を Java VM が自動的に行う明示的メモリ管理の半自動適用方法を提案する．

3.1 システム概要

明示的メモリ管理の半適用技術を適用したシステム概要を**図 1** に示す．明示的メモリ管理を有する Java VM（JVM-EH）の起動時に，Explicit メモリ生成点を指定した外部指定ファイルを読み込む (1)．Java プログラム実行時に必要となったバイトコードは逐次 JVM-EH が読み込む (2)．Explicit メモリ生成点が読み込んだバイトコードに含まれている場合，その指定点を Explicit メモリを生成するように変換する (3)．JVM-EH は変換したバイトコードを実行し (4)，Explicit メモリ生成点でメモリ空間から Explicit メモリ領域を確保する (5)．また，JVM-EH は Explicit メモリ領域への参照の数を監視し，その数が閾値を下回ったとき，その Explicit メモリを解放する．ただし，解放領域内に live オブジェクトが存在した場合，JVM-EH は新たな Explicit メモリを生成し，その領域に live オブジェクトを移動し，継続して管理する (5)．また，Explicit メモリを適用しないオブジェクトは，

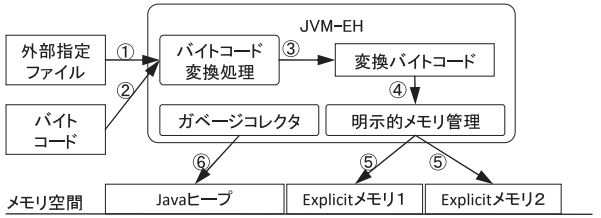


図 1　提案システムの概要
Fig. 1　Organization of the proposed system.

従来どおり Java ヒープに配置し，ガベージコレクタが管理する (6)．

3.2 明示的メモリ領域の生成とオブジェクトの配置

Explicit メモリの生成とその領域へ配置するオブジェクトの指定は，その適用点を外部から指定する．外部から指定する内容として，Explicit メモリの生成 API の挿入点と Context 型/Reference 型それぞれの制御 API [⋆1]がある．Explicit メモリは，その領域へのオブジェクト配置が指定される前に生成されていればよい．そのため，Context 型の適用区間あるいは，Reference 型の基点オブジェクトのみを外部から指定するものとし，その指定点の直前に Explicit メモリを生成するようバイトコードを修正する．

Context 型のオブジェクト配置方法を外部から指定する場合，enter/exit メソッドの挿入場所を指定する．外部から enter/exit メソッドで挟む区間を指定する場合，その区間をバイトコードから特定できる必要がある．任意の場所を区間指定するには，ソースコードの行番号を外部指定することが考えられるが，クラスファイルに行番号が保存されていない場合がある．この場合，行番号に対応するバイトコードの位置を特定することができない．また，Java プログラムは目的の処理に対してメソッドを定義することが一般的になっており，処理を区間指定する Context 型において区間指定対象をメソッドに限定することで実用性を低下させる場合が少ないと考える．

以上のことから，外部から指定する区間を，メソッド呼び出し点の前後とする．すなわち，外部から指定できる区間とは呼び出し先のメソッドの先頭から末尾までである．指定内容は，区間指定対象メソッドの呼び出し点を含むメソッドと，区間指定対象メソッドの組みである．

⋆1 Context 型の場合は適用区間の指示，Reference 型の場合は基点オブジェクトの指示．

修正前	修正後
public class sample.Test1 public void caller(); Code: … aload_0 invokespecial　callee() … return	public class sample.Test1 public void caller(); Code: … new　ContextExplicitMemory invokespecial　ContextExplicitMemory.<init>() astore_1 aload_1 invokevirtual　ContextExplicitMemory.enter() aload_0 invokespecial　callee() aload_1 invokevirtual　ContextExplicitMemory.exit() … return

図 2　Context 型 Explicit メモリを利用する場所のバイトコード修正処理
Fig. 2　Bytecode transformation for context-type ExplicitMemory.

修正前	修正後
public class sample.Test2 public void caller(); Code: … new　data.BigData invokespecial　data.BigData.<init>() putfield … return	public class sample.Test2 public void caller(); Code: … new　ReferenceExplicitMemory invokespecial　ReferenceExplicitMemory.<init>() astore_1 aload_0 aload_1 ldc_w　data.BigData invokevirtual　ReferenceExplicitMemory .newInstance(Class) checkcast　data.BigData putfield … return

図 3　Reference 型 Explicit メモリを利用する場所のバイトコード修正処理
Fig. 3　Bytecode transformation for reference-type ExplicitMemory.

以下に区間指定対象メソッド（TARGET）を sample.Test1.callee メソッド，そのメソッドの呼び出し点を含むメソッド（LOCALE）を sample.Test1.caller メソッドとした場合の外部指定内容の例を示す．

LOCALE=⟩sample.Test1.caller(), TARGET=⟩sample.Test1.callee()

Java VM が LOCALE で指定されたメソッド定義を読み込む場合，バイトコードに対するパターンマッチングにより TARGET の呼び出しを検出し，それを enter/exit メソッドの呼び出しで囲み，enter メソッドの直前に Explicit メモリを生成するようバイトコードを修正する．修正前後のバイトコードイメージを**図 2** に示す．左側が修正前のバイトコードであり，右側が修正後のバイトコードのイメージである．

Reference 型のオブジェクト配置方法を外部から指定する場合，基点オブジェクトの生成点を指定する．指定内容は，基点オブジェクトを生成しているメソッド名（sample.Test2.caller）とそのオブジェクトのクラス（data.BigData）とする．

LOCALE=⟩sample.Test2.caller(), TARGET=⟩data.BigData

Java VM がバイトコードを読み込む際，LOCALE で指定したメソッドが存在する場合，TARGET で指定されたオブジェクトの生成点を基点オブジェクトとして生成するようバイトコードを修正する．修正前後のバイトコードイメージを**図 3** に示す．左側が修正前のバイトコードであり，右側が修正後のバイトコードのイメージである．

オブジェクト配置方法として Context 型と Reference 型のいずれにおいても，指定する LOCALE 内に複数の TARGET が存在する場合，それぞれの TARGET に対して Explicit メモリを生成する．

3.3　明示的メモリ領域の解放

Explicit メモリ領域の解放は，その領域内に配置されたオブジェクトの多くが dead になった時点で実施することが望ましい．しかし，2.2 節で述べたとおり，大規模なプログラムにおいて，その dead/live 判定が困難な場合がある．そこで，ソースコードを修正せずに明示的メモリ管理を適用するため，Explicit メモリ領域を自動的に解放する方法を採用する．

理想的には，Java VM が Explicit メモリ領域内のすべてのオブジェクトについて dead/live 判定し，live オブジェクトが閾値を下回ったときに領域を解放する方式が考えられる．しかし，この処理は Mark-Compact GC 手法における Mark 処理と同等の処理時間を要する．Mark 処理は Mark-Compact GC 処理時間の半分程度を要することが経験的に分かっているため，Mark 処理による dead/live 判定には多くの処理時間を要することが容易に想像できる．

そこで，Explicit メモリ領域の解放を高速化するために導入した ExplicitHeapReferenceArray（EHRA）[2] を応用し，Explicit メモリ領域内の live オブジェクト数を概算する方法を提案する．EHRA は Java ヒープ領域および Explicit メモリを一定サイズのメモリブロックに分割し，個々のブロックの参照先が，ある Explicit メモリ領域に存在する場合，参照先の Explicit メモリを一意に特定する ID 番号を保持する配列構造である（**図 4**）．

たとえば，Java ヒープ内のオブジェクト（obj）が ID 番号 1 の Explicit メモリ内のオブジェクトを参照している場合，obj に対応する EHRA の要素の値が ID 番号 1 となる．また，EHRA の 1 要素に対応するメモリブロック内から異なる Explicit メモリ領域への参照

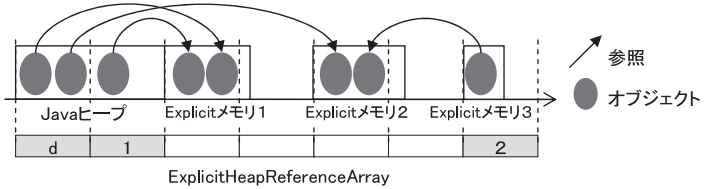


図 4　オブジェクトの参照関係と ExplicitHeapReferenceArray の対応関係
Fig. 4　Relation between object references and ExplicitHeapReferenceArray.

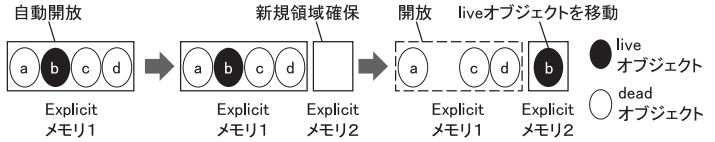


図 5　Explicit メモリの自動解放にともなう live オブジェクトの新規 Explicit メモリへの移動
Fig. 5　Live object migration from reclaimed ExplicitMemory to new ExcplicitMemory.

がある場合，その EHRA の要素は d（dirty）となる．したがって，特定の ID 番号を格納している EHRA の要素数を調べることで，特定の Explicit メモリ領域への参照数を概算できる．概算した被参照数が少ない Explicit メモリを live オブジェクトの少ない領域と考え，その数が閾値を下回ったときにその領域を解放する．図 4 の例において，閾値を 1 とした場合，ID 番号 3 の Explicit メモリが参照されていることを示す EHRA の要素数が 0 個であるため，これらの領域を自動解放対象とする．

この閾値を固定値として指定することもできる．しかし，ここで用いる参照数は概算値であるため，アプリケーションによっては概算精度が悪くなることも考えられる．そこで，本論文では固定の閾値で解放対象とする Explicit メモリを選択せず，Explicit メモリへ移動してきたオブジェクトの合計サイズを超えるまで，参照数の概算値が少ない順に Explicit メモリを選択し，解放対象にする．

この自動解放処理は，メモリ領域が一杯になったときに Explicit メモリを解放することが考えられる．これは自動解放処理の回数を少なくできるため，実行性能を向上させることができる可能性がある．しかし，Explicit メモリの解放は安全性を確認する処理を実施する．この処理は解放領域外のオブジェクトから解放領域内のオブジェクトへの参照の有無を確認するため，解放対象外のメモリ領域が大きいほど処理時間を多く要する．このことから，本論文では自動解放処理は Minor GC のたびに実施し積極的に解放を試みる．

また，Explicit メモリに配置されるオブジェクトは，システム起動から終了まで live な長寿命オブジェクトか，Minor GC では回収できず，Major GC で回収対象となるような中寿命なオブジェクトである[⋆1]．前者のような長寿命オブジェクトを Explicit メモリに配置した場合，多数の領域から参照されることが予想される．そのため，EHRA を導入した参照数の概算数も多くなり，解放対象領域に選ばれにくくなる．後者のような中寿命オブジェクトは，必要な処理終了後には不要となり，回収可能になるはずである．したがって，自動解放時に，その領域内の live オブジェクトは，いずれ dead になることが期待できることから，継続して Explicit メモリ領域で管理する．すなわち，解放対象の Explicit メモリ内に存在する live オブジェクトの移動先を新たに確保した Explicit メモリとする（**図 5**）．

EHRA はメモリ領域間の参照関係を管理するため，Explicit メモリは ID 番号を持っている．自動解放を実施した Explicit メモリ内にオブジェクトが残存していると，そのオブジェクトは新たな Explicit メモリに移動するため Explicit メモリの合計数は減少しない．この状態で Explicit メモリの生成が続くと，Explicit メモリの合計数は増加し続け，ID 番号が枯渇する可能性がある．この場合，使用サイズの小さな Explicit メモリを選択し，それらを同一 ID 番号に設定する．これにより複数あった Explicit メモリは論理的に 1 つに統合できるため，ID 番号の枯渇を阻止することができる．ただし，この処理により EHRA で管理する領域間の参照関係が崩れるため，統合する前の ID 番号を統合した後の ID 番号に書き換える必要がある．たとえば，ID 番号が 1，2，3 の Explicit メモリを統合して ID 番号を 4 にする場合，EHRA で 1，2，3 の要素を 4 に書き換える．

また，自動解放を実施した結果オブジェクトを回収できない場合，次の自動解放の際に再び同じ Explicit メモリが選択されてしまう可能性がある．この場合，1 度選択して自動解放を実施した Explicit メモリは，一定回数だけ自動解放対象の Explicit メモリとして選択されないなどの対策が考えられる．しかし，4 章で実施した 5 種類の評価プログラムでは Explicit メモリ内のオブジェクトが解放できず，同一の Explicit メモリを選択し続けるような現象は発生しなかった．今後，様々なテストプログラムに提案手法を適用する中で，本問題が発生したときに検討する予定である．

⋆1 短寿命なオブジェクトは Explicit メモリへ移動する前に Minor GC で回収される．

4. 評　　価

本章では，提案する明示的メモリ管理の半自動適用技術評価結果について述べる．評価で用いた Java VM は，HotSpot VM（JDK5.0 update 5）に我々の提案する明示的メモリ管理を適用した Java VM（以下，JVM-EH）と，適用しない Java VM（以下，JVM）である．

4.1 実 験 環 境

実験に用いるマシン環境は，CPU：Xeon（1.6 GHz，Quad Core×2），メモリ：6 GB，OS：Linux（CentOS4.7）である．評価プログラムとその説明を**表 1** ⋆1に示す．明示的メモリ管理の効果を評価するため，Major GC が発生するプログラムを選定した．

JVM のヒープサイズは JVM-EH における Java ヒープサイズと Explicit メモリサイズの合計とする．各評価プログラムに対する外部指定内容を**表 2** ⋆2に示す．

xml.validation および Terasoluna は Context 型 Explicit メモリを適用し，Tomcat [6] および Hibernate [7] は Reference 型 Explicit メモリを適用し，SPECjbb2005 [4] においては両方を適用している．

4.2 明示的メモリ管理の適用効果

各プログラムにおける Major GC，Minor GC および JVM-EH における Explicit メモリの自動解放による平均停止時間を**表 3** に示す．JVM-EH ではすべてのプログラムにおいて Major GC の発生を抑止できている．Explicit メモリの自動解放処理時間は，Hibernate と SPECjbb2005 を除いて Major GC 時間の 10 分の 1 以下である．また Minor GC 時間についてもほぼ同じ停止時間となっている．性能評価結果を**表 4** に示す．バイトコード（BC）変換箇所とは実際に BC を変換した数である．BC 変換による性能比とは，BC 変換前のプ

表 1　評価プログラムと Java VM のメモリ設定
Table 1　Benchmark programs and their memory settings.

Program	Java ヒープ	Explicit メモリ	概要
xml.validation	768 MB	256 MB	XML 構文検証（SPECjvm2008 [4]）.
SPECjbb2005	512 MB	512 MB	模擬電子商取引（Warehouse=8）
Terasoluna	896 MB	128 MB	バッチ処理フレームワーク.
Tomcat	896 MB	128 MB	Web アプリケーションサーバ.
Hibernate	768 MB	256 MB	O/R マッピングフレームワーク.

Terasoluna は Terasoluna Batch Framework [5] を利用する．

⋆1 Terasoluna はデータベースアクセス機能を用いたプログラムを実行し，Tomcat はセッションごとに XML をパースし保持するアプリケーションを実行し，Hibernate はデータベースを検索するアプリケーションを実行する．

⋆2 外部指定内容のクラス名やメソッド名はパッケージ名を含める必要があるが，ここでは見やすさのため，パッケージ内で唯一に特定できる場合は省略している．また，メソッド名については多重定義がなくソースコード中で一意に特定できる場合は，その引数の記述も省略している．

表 2　外部指定内容
Table 2　Specification for bytecode transformation.

Program	LOCALE	TARGET
xml.validation	Main.harnessMain()	Main.executeWorkload()
SPECjbb2005	JBBmain.run()	TransactionManager.go()
	JBBmain.doItForValidation()	Company.<init>()
	JBBmain.doIt()	Company.<init>()
	Infrastructure.createSortedStrage()	MapDataStrage
	Infrastructure.createSortedStrage()	TreeMapDataStrage
Terasoluna	StandardBLogicExecutor.executeBLogic()	BLogic.execute()
Tomcat	StandardSession.init<>()	java.util.Hashtable
Hibernate	ThreadLocalSessionContext.doBind()	java.util.HashMap
	Configuration.buildSessionFactory()	SessionFactoryImpl

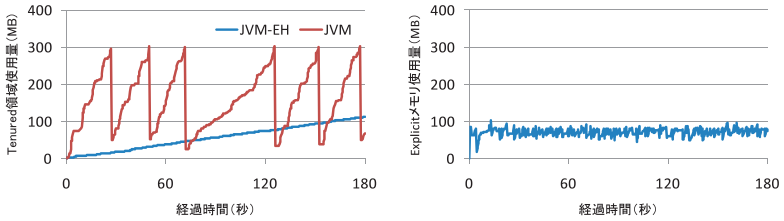
表 3　メモリ管理による平均停止時間
Table 3　Pause-time caused by memory management.

	Major GC［秒（回数）］		Minor GC［秒］		自動解放［秒］
Program	JVM	JVM-EH	JVM	JVM-EH	JVM-EH
xml.validation	0.50(7)	—(0)	0.03	0.04	0.03
SPECjbb2005	1.47(34)	—(0)	0.15	0.16	0.40
Terasoluna	0.89(3)	—(0)	0.21	0.18	0.03
Tomcat	0.92(18)	—(0)	0.03	0.03	0.00
Hibernate	0.45(10)	—(0)	0.06	0.07	0.12

表 4　性能評価結果
Table 4　Results of performance evaluations.

Program	BC 変換箇所	BC 変換による性能比（%）	実行性能比（%）
xml.validation	1	99.3	72.1
SPECjbb2005	4	99.9	55.3
Terasoluna	1	99.7	99.9
Tomcat	1	99.9	99.1
Hibernate	2	99.5	99.4

*BC：バイトコード



(a) Tenured area.　(b) ExplicitMemory.

図 6　Hibernate のメモリ使用量
Fig. 6　Memory usage of Hibernate.

ログラムに対する BC 変換による性能比である．ただし BC 変換により明示的メモリ管理 API を挿入するが，明示的メモリ管理は利用しない．BC 変換によるプログラムの実行性能低下は 1%未満である．

しかし，xml.validation および SPECjbb2005 においては実行性能が大きく劣化している．これは，Context 型 Explicit メモリの性能が良くないことに起因している[2]．Terasoluna も Context 型 Explicit メモリを適用しているが，Terasoluna では Major GC 時間がアプリケーション実行時間に占める割合が大きいため，明示的メモリ管理を適用することで性能が向上する．そのため，Context 型 Explicit メモリの適用による性能劣化の影響が隠ぺいされている．

Hibernate におけるメモリ使用量の推移を**図 6** (a) に示す．Explicit メモリの使用量の推移を図 6 (b) に示す．JVM は Tenured 領域の使用量が徐々に増えていき，500 MB 付近で Major GC が発生し，領域が解放されている．それに比べ，JVM-EH では Tenured 領域の使用量の増加がなだらかなため，Major GC の発生頻度を削減できている．また，Explicit メモリの使用量は 100 MB 付近で安定して利用しており，メモリを効率的に利用できている．これ以外のプログラムについても同様の傾向を示すことを確認している．これは提案する半自動適用技術により，Explicit メモリを従来どおり利用できていることを示している．

また，ここで述べた評価結果は提案する明示メモリ管理の半自動適用技術を適用せずソースコードを修正し Explicit メモリを利用する場合の性能とほぼ同じである．

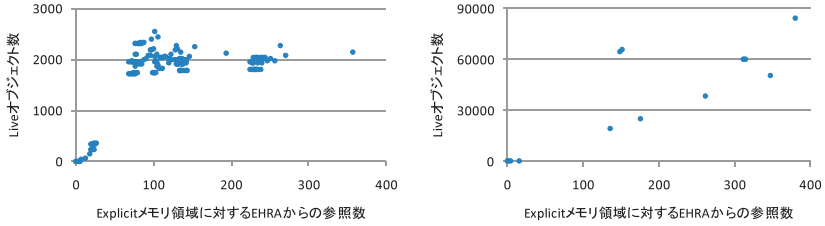
4.3　EHRA による Explicit メモリの解放

表 5 に EHRA の全要素数に対する Dirty である要素数の割合を示す．

xml.validation，SPECjbb2005 および Hibernate の Dirty 率が比較的高い．EHRA の要素が Dirty であれば，live オブジェクトを特定するためにはブロック内のオブジェクトの参照先を 1 つずつ調べる必要がある．これが自動解放処理時間（表 4）が長くなっている要因の 1 つと考えられる．

表 5　EHRA の Dirty 率
Table 5　Dirty ratio of EHRA.

Program	Dirty（%）
xml.validation	5.3
SPECjbb2005	47.4
Terasoluna	0.0
Tomcat	0.1
Hibernate	1.7



(a) xml.validation.　(b) Hibernate.

図 7　EHRA の要素による参照数の概算値と live オブジェクト数の関係
Fig. 7　Relation between number of references of EHRA and number of live objects.

EHRA から Explicit メモリを参照する数と live オブジェクトの数の関係を示したグラフを**図 7** に示す．

xml.validation は EHRA から Explicit メモリを参照する数と Explicit メモリ内の live オブジェクト数との間の相関関係が弱い．それに対して，Hibernate では比較的強い相関がみられる．提案手法では EHRA からの参照数を live オブジェクトの概算値として Explicit メモリを解放するか否かを決定しているため，xml.validation では live オブジェクトの多く含まれる Explicit メモリを解放してしまっているものと考えられる．これに対して，Hibernate では live オブジェクト数の概算精度が高く，Explicit メモリを効果的に解放できる．SPECjbb2005 は xml.validation と同様の傾向を示しており，その他のプログラムは Hibernate と同様の傾向を示していることを確認している．

5. 関 連 研 究

Java において，GC による停止時間を短縮する方法として，GC 手法を改良する方法と，GC 対象外メモリを利用する 2 つの手法が主に研究されている．

GC によって生じる停止時間を短縮する手法として Concurrent Mark-Sweep GC（CMS）[8] 手法や，Real-Time GC 手法が実用化されている．これらの手法は，ミューテータと並行してコレクタを実行するため，プログラムの実行性能が大幅に低下するという問題がある．また，CMS では Tenured 領域の断片化が進むと，Major GC が発生し長時間の停止を免れない．Major GC を発生させにくくした Garbage First Garbage Collection [9] も提案されているが，実行性能が低下してしまうという問題は残っている．

GC 対象外メモリを利用する技術として，Real-Time Specification for Java [10),11)]（以下，RTSJ）で定められている Scoped Memory がある．Scoped Memory では，一連の処理で生成するデータオブジェクトを明示的に確保した Scoped Memory 内に生成することで，Java ヒープメモリの使用量を削減し GC を発生しにくくすることができる．一連の処理が終了すると Scoped Memory 内部に配置されたオブジェクトごとその領域を解放する．Scoped Memory では領域解放を可能にするため，オブジェクトの参照関係に関する制約をプログラマが保証する必要がある．この制約のため，既存のプログラムへ Scoped Memory を適用するのは容易でない．また，この制約はプログラム実行時にチェックする必要があるため，プログラムによっては実行性能が大きく低下してしまうという問題もある[12),⋆1]．それに対して，明示的メモリ管理を適用した場合，性能の低下を少なく抑えることができる．明示的メモリ管理は一般的な Java 言語の仕様に沿って記述できるが，提案する半自動適用技術を導入することでソースコードを修正することなく明示的メモリ管理を適用できる．

明示的メモリ管理は複数のメモリ領域を利用することから，多世代型の GC [13),14)] に似ている．多世代型 GC では Java ヒープを世代という概念で複数の領域に分割する．オブジェクトは GC 経過回数を age として記録し，その age に応じて，対応する世代領域に移動して管理される．すなわち，ガベージコレクタはオブジェクトの age を基準に管理している．それに対し，本論文で提案する明示的メモリ領域の自動解放方法は，ユーザ指定によって関連したデータを 1 つの領域に配置する．その領域の管理は配置したオブジェクトへの参照の有無を基準にする点で異なる．

ソースコードを修正せずにプログラムの動作を変更する方法として，Aspect Oriented Programming（AOP）[15] がある．AOP は複数クラスに横断的な処理を分離し，Aspect として定義する．Aspect はソースコード外から指定された場所に挿入される．これによりソースコードを修正せずに処理を追加することができる．Java における AOP の実現方法は，コンパイル時に静的に Aspect を挿入しバイトコードを生成する方法や，バイトコードを読み込むときやプログラム実行時などに動的に Aspect を挿入する方法がある．AOP は指定した処理に対して Aspect を追加する技術である．本論文で提案するバイトコード書き換えは，AOP と同様に処理を挿入するが，それに加えて置換処理も行う点で異なる．

6. ま と め

本論文では，長時間 GC によるアプリケーション停止回数を削減する明示的メモリ管理 API の適用をソースコード修正なしで可能とする技術を提案し，大規模オープンソースミドルウェアなどを対象として評価した結果について示した．

提案手法では，明示的メモリ領域の生成点とその領域へ生成するオブジェクトを外部から指定し，その指定点を含むバイトコードに対し，明示的メモリ管理を利用するよう Java VM が修正する．明示的メモリ領域の解放は，領域内の live オブジェクトが減少したことを検出し自動的に Java VM が解放する．解放にともない，live オブジェクトを新たに生成した Explicit メモリへ移動することで，継続して live オブジェクトを管理し，Tenured 領域の使用量増加を抑止する．

提案する明示的メモリ管理の半自動適用技術をベンチマークプログラムやオープンソースのミドルウェアに適用した結果，Tenured 領域へのオブジェクトの移動を削減でき，Major GC の発生回数を抑止できていることを確認した．また，Reference 型 Explicit メモリの半自動適用による実行性能の低下は 1%程度であった．また，自動解放処理は Minor GC 処理時間程度で完了するため，最大停止時間を大幅に削減できる見込みがあり，提案手法の有用性を確認した．

Context 型 Explicit メモリを適用した xml.validation と SPECjbb2005 の場合，それ自身のオーバヘッドと EHRA による live オブジェクトの概算精度低下が，SPECjbb2005 および xml.validation における性能低下の主要因であると考えられる．これらの改善については今後の課題である．

⋆1 Scoped Memory を利用する場合，領域を解放しても不正な参照が発生しないことをプログラマが保証する必要がある．領域間の参照関係は実行中にチェックされるため，実行性能が大きく低下する．

参 考 文 献

1) Gosling, J., Joy, B., Steele, G.L. and Bracha, G.: *The Java Language Specification*, Addison Wesley (2005).
2) 小幡元樹，西山博泰，足立昌彦，岡田浩一，長瀬卓真，中島　恵：Java における明示的メモリ管理，情報処理学会論文誌，Vol.50, No.7, pp.1693–1715 (2009).
3) Bak, L., Duimovich, J. and Fang, J.: The New Crop of Java Virtual Machines, *Proc. 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (1998).
4) Standard Performance Evaluation Corporation (2009). http://www.spec.org
5) Terasoluna フレームワーク (2009). http://terasoluna.sourceforge.jp
6) Apache Tomcat (2009). http://tomcat.apache.org
7) Hibernate (2009). http://www.hibernate.org
8) Printezis, T. and Detlefs, D.: A generational mostly-concurrent garbage collector, *Proc. 2nd International Symposium on Memory Management ISMM '00*, Vol.36, No.1 (2000).
9) Detlefs, D., Flood, C., Heller, S. and Printezis, T.: Garbage-Fist Garbage Collection, *Proc. International Symposium on Memory Management ISMM '04* (2004).
10) Corsaro, A. and Cytron, R.K.: Efficient Memory-Reference Checks for Real-time Java, *Proc. 2003 Conference on Languages, Compilers, and Tools for Embedded Systems* (2003).
11) Pizlo, F., Fox, J.M., Holmes, D. and Vitek, J.: Real-Time Java Scoped Memory: Design Patterns and Semantics, *Proc. 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (2004).
12) Beebee, W.S. and Rinard, M.C.: An Implementation of Scoped Memory for Real-Time Java, *Proc. Embedded Software, 1st International Workshop* (2001).
13) Jones, R. and Lins, R.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley (1996).
14) Stefanovic, D., Hertz, M., Blackburn, S.M., McKinley, K.S. and Moss, J.E.B.: Older-first Garbage Collection in Practice: Evaluation in a Java Virtual Machine, *Proc. 2002 Workshop on Memory System Performance*, pp.25–36 (2002).
15) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An Overview of AspectJ, *European Conference on Object-Oriented Programming 2001*, pp.327–353 (2004).

(平成 21 年 9 月 29 日受付)
(平成 22 年 1 月 5 日採録)

足立　昌彦

2006 年神戸大学大学院修士課程修了．同年（株）日立製作所入社．同社システム開発研究所にて Java に関する研究開発に従事．

小幡　元樹（正会員）

1973 年生．2003 年早稲田大学大学院理工学研究科博士課程修了．工学博士．（株）日立製作所システム開発研究所主任研究員．Java に関する研究開発に従事．

西山　博泰（正会員）

1993 年筑波大学大学院工学研究科博士課程修了．工学博士．（株）日立製作所システム開発研究所主任研究員．最適化コンパイラ，Java 実行環境等言語処理系の研究に従事．ACM，IEEE 各会員．

岡田　浩一

1979 年生．2003 年信州大学大学院工学系研究科修士課程修了．同年（株）日立製作所入社．ソフトウェア事業部にて Java 仮想マシンに関する製品開発に従事．

長瀬　卓真

1981 年生．2006 年法政大学大学院情報科学研究科修士課程修了．同年（株）日立製作所入社．ソフトウェア事業部にて Java 仮想マシンに関する製品開発に従事．

中島　　恵

1966 年生．東海大学情報数理学科卒業．（株）日立製作所ソフトウェア事業部担当部長．1989 年入社以来，C/C++，Fortran コンパイラの製品設計，開発，および Java 仮想マシンの製品計画，開発を経て，2007 年よりアプリケーションサーバの製品計画，開発とりまとめに従事．