Vol.2014-ARC-212 No.1
2014/10/6

GC実行時の高速なコンパクションを可能にするハードウェア支援手法の検討

井手上 慶[1]　河村 慎二[1]　津邑 公暁[1,a)]

概要：スマートフォンなどの普及に伴い，ガベージコレクション（GC）の性能が与える影響範囲が拡大している．一方，GCは主にアルゴリズム面で改良がなされてきたが，GC実行時のレスポンス低下など，重要な問題の根本的解決には未だ至っていない．これに対し我々は，ハードウェア支援によりGCを高速化する手法をこれまでにいくつか提案しており，その有用性について検討してきた．本稿では，まず我々が提案している二つの手法を取り上げ，それぞれ評価結果を示すとともにその有用性について述べる．これらの手法はいずれも，GCにおける基本的な構成処理要素に着目し，その高速化を図るものである．その後，現在我々が取り組んでいるハードウェア支援を用いたコンパクション機能について述べる．コンパクション機能を実装している既存のGCアルゴリズムはいくつか存在しているが，オブジェクトの移動時には当該オブジェクトを参照しているポインタを張り替える必要があり，これは一般にコストが比較的大きい．そこで本手法では，オブジェクト間の参照関係を記憶する専用の表をプロセッサに追加し，これを利用することで高速なポインタの書き換え，およびコンパクション機能の実現を目指す．そして最後に，この手法により期待される効果について考察する．

1. はじめに

スマートフォンなど，一般にメモリ容量が小さく，メモリ管理システムの重要性が非常に高いモバイル機器の普及に伴い，**ガベージ・コレクション（Garbage Collection: GC）**の性能が与える影響範囲が拡大している．一方，このGCに関しては古くから，サーバサイドJava環境等において全体性能に大きな影響を与えることが知られており，GC実行時のプロセス全停止によるレスポンス低下などが問題視されてきた．これに対し，アプリケーションとGCを並行動作させることで，GCの実行に伴うシステムの最大停止時間を緩和するConcurrent GC[1]などが提案されている．しかし，Concurrent GCでは並行動作によってGC処理のスループットが犠牲となっている問題があり，高いスループットが求められるシステムには適していない．さらに，スループットの低下はエネルギー消費の悪化にも繋がるものであり，特にバッテリ駆動が前提となるモバイルシステムにとっては重大な問題となり得る．そのため，特にモバイルシステムの性能を改善するには，GCの実行による最大停止時間を緩和するだけでなく，GC自体の実行時間を削減し，システム全体のスループットを高く保つことが重要となる．

さて，従来GCの高速化は，主にアルゴリズムの改良という観点から研究されてきた．しかし上述したように，Concurrent GCでは並行動作によってGC処理のスループットが犠牲となるなど，GCの抱える重要な問題の根本的解決には未だ至っていない．よって，GCによる性能悪化を緩和するために，システムや各種パラメータのチューニングによって，これを補ってきたというのが実情である．

これに対し，我々はこれまでにハードウェア的にGCの実行をアシスト可能なプロセッサ構成方式[2],[3]を提案し，その有用性について検討してきた．本稿では，まずこれら二つの手法の概要を示し，その評価結果および有用性について述べる．そして，その後現在取り組んでいる新たな手法を紹介する．我々は現在，GC実行時に生きているオブジェクト間の全ての参照がトレースされる必要がある点に着目し，その参照関係を専用の表に記憶しておくことで，高速なコンパクション機能を実現する手法について研究している．メモリのコンパクションを高速に実現することで，フラグメンテーションの発生を抑制し，メモリ利用効率の向上が期待できる．

2. 研究背景

本章では，まずガベージ・コレクションについて概説し，

1 名古屋工業大学
Nagoya Institute of Technology, Japan
a) tsumura@computer.org

© 2014 Information Processing Society of Japan

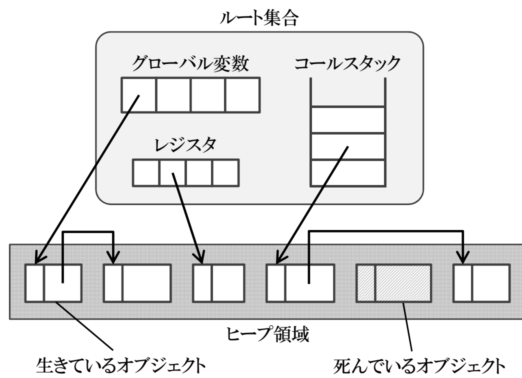


図 1　プログラム実行時のヒープ領域と参照関係の様子

その後ハードウェア支援による GC 高速化の既存研究について述べる．

2.1　ガベージ・コレクション

ガベージ・コレクション（GC）とは，プログラム実行のために動的に確保したメモリ領域のうち，ヒープ領域内の不要となった領域を自動的に解放する機能である．ここで，プログラム実行時のヒープ領域，およびオブジェクト間の参照関係の様子の例を**図 1** に示す．ヒープ領域内に生成されたオブジェクトへのポインタはグローバル変数やコールスタック，レジスタ等の，アプリケーションから直接参照可能な領域に格納される．このような領域の集合を**ルート集合**と呼ぶ．そしてヒープ領域内のオブジェクトのうち，ルート集合を起点として参照可能，つまりアプリケーションから参照可能なものを，**生きているオブジェクト**と呼ぶ．逆にルート集合から参照不能なオブジェクトを，**死んでいるオブジェクト**と呼び，GC はこのようなオブジェクトに割り当てられたメモリ領域を不要な領域として解放する．なお，オブジェクトにはルート集合から参照されているものだけでなく他のオブジェクトから参照されているものも存在する．こうした他のオブジェクトから参照されているオブジェクトを**子オブジェクト**と呼ぶ．

GC の代表的なアルゴリズムの一つに **Mark & Sweep**[4] がある．このアルゴリズムは，Mark フェーズと Sweep フェーズという 2 つのフェーズで構成される．まず Mark フェーズでは，ルート集合からポインタを辿り，生きている全てのオブジェクトにマークを付ける．その後 Sweep フェーズへと移行してヒープ領域全体を走査し，Mark フェーズにおいてマークの付けられなかったオブジェクト，つまり，ルート集合を起点として参照不能な死んでいるオブジェクトに割り当てられたメモリ領域を不要な領域として解放する．Mark & Sweep はこの 2 つのフェーズを交互に繰り返しながら動作する．

また，その他の代表的な GC アルゴリズムとして，Copying[5] と Reference Counting[6] があるが，現在研究されている全ての GC アルゴリズムは，これら 3 つのアルゴリズムの組み合わせ，もしくは改良であることが知られている [7]．特に Mark & Sweep は実装が比較的容易であることから，他のアルゴリズムとの組み合わせが数多く存在する．

2.2　既存研究

1 章で述べたとおり，GC は主にアルゴリズムの改良という観点で研究されてきた．一方，ハードウェアによる GC 支援も，わずかながら既存研究が存在する．本節では，その代表例として SILENT[8] と Network Attached Processing(NAP)[9] について取り上げる．

SILENT は，NUE プロジェクトによって開発された Lisp 専用マシンである．SILENT では GC アルゴリズムに Mark & Sweep をベースとした ConcurrentGC を採用しており，GC プロセスとアプリケーションが並行に動作する．しかし，GC プロセスがマークフェーズを実行中にアプリケーションがオブジェクトの参照を書き換えた場合，生きているオブジェクトへのマーク漏れが発生する可能性がある．そこで SILENT ではライトバリアと呼ばれる，書き込みを検知し，その書き込みによるデータ不整合を防ぐための同期処理を用いて，この問題に対処している．

SILENT では，アプリケーションによるポインタの書き変えをライトバリアによって検知し，それを GC プロセスに知らせることでマーク漏れを防いでいる．具体的には，マーク済みオブジェクトが未マークオブジェクトを参照するようにポインタが書き換えられた際，アプリケーションはその参照元オブジェクトを GC プロセスに通知する．GC プロセスは通知されたオブジェクトをルート集合と見なし，再度マークを行いポインタの書き換えによるマーク漏れを防ぐ．SILENT では，このライトバリアをマイクロプログラムで記述されたサブルーチンとして実装することで GC 実行を高速化している．この高速なライトバリアによって，SILENT における GC の停止時間は最大で 100 マイクロ秒以下と非常に短時間に抑えられている．

もう一つの既存研究である NAP は，Azul Systems 社が開発した，Java の実行に特化したフレームワークである．NAP では PauselessGC という，Mark & Sweep と Copying を組み合わせた ConcurrentGC を改良したアルゴリズムが採用されている．そのため，SILENT と同様に，生きているオブジェクトへのマーク漏れが発生する可能性がある．そこで NAP では，リードバリアと呼ばれるオブジェクトを読み出す際に行う同期処理を用いてマーク漏れを防いでいる．

NAP では，ポインタが参照しているオブジェクトをアプリケーションがロードする度，そのポインタが GC によって巡回済みかどうかをリードバリアによってチェックし，もし未巡回であればそのポインタを GC スレッドに通知し，マーク漏れを防ぐ．このリードバリアは，ポインタ

が GC によって巡回済みかをチェックする特殊なロード命令を新たに実装することで実現されている．ポインタのチェックに要するコストは，通常のロード命令のサイクル数と比較して 1 サイクル多い程度であるため，非常に高速にリードバリアを実行できる．

しかし，これら既存手法はいずれも特定言語における GC 実装で必要となるバリア同期のみを高速化するものである．これに対して我々は代表的 GC アルゴリズムに共通して必要となる処理自体を高速化するという，これらの既存手法と異なる観点から GC の高速化を目指している．GC の基本的な処理をハードウェア支援することにより，多くの GC アルゴリズムの高速化を目指す．さらに，ハードウェア支援により GC の高速化をソフト・ハードウェアの協調問題として発展させ，チューニングに頼らずとも，ユーザがシステムの性能を引き出せるようになることも期待できる．

3. コールスタック上のポインタ判別高速化

前章で述べたとおり，我々はハードウェア支援による GC の高速化を目指しており，これを実現する手法をいくつか提案している．その一つに，コールスタック上のポインタ判別に着目した手法 [2] がある．本章では，まずその概要を述べ，その後評価結果について考察する．

3.1 ルート集合からのポインタ探索

Mark & Sweep などの多くの GC アルゴリズムでは，ルート集合からポインタを探索してオブジェクトを辿るという共通の動作が存在する．このルート集合の一つであるコールスタック上の各フレームには，ローカル変数やリターンアドレス，関数に与えられた引数などが格納されている．しかしローカル変数には，オブジェクトへのポインタである参照型変数だけでなく， int などのプリミティブタイプ型変数も存在しており，これらは区別無くコールスタック上に格納されている．そのため，GC 実行時にコールスタックを起点としてポインタを探索する際には，コールスタック上の値の中からポインタを判別する処理が必要である．

例えば，JavaVM の一種である HotspotVM[10] では，コールスタック内でポインタが格納されている位置を**スタックマップ**と呼ばれるビット列で管理している．Java のバイトコードの場合，ローカル変数などへ値を格納する命令は，格納する値の型に応じて異なっている．そこで，実行する命令の種別をもとにスタックマップを作成することで，コールスタック上で参照型変数が格納されている位置を特定できる．しかし，条件分岐などによって実行フローが変化することから，このスタックマップは GC 実行時に各実行フローに合わせて適宜作成する必要があり，これに要するコストは GC 処理のオーバヘッドとなり得る．

表 1 シミュレーション対象となるプロセッサの構成

マシン	ARM-RealView PBX
プロセッサ	ARMv7
周波数	2.0 GHz
メモリ	128MB
OS	Linux 2.6.38.8-gem5

3.2 専用表によるポインタ管理

上述したポインタ判別に要するオーバヘッドを削減するために，提案する手法ではコールスタック上のポインタを管理する専用表をプロセッサに追加して利用する．なお専用表は，一次登録表，二次登録表と呼ばれる二つの表を追加し，これらの表はカレントフレーム内のポインタ，カレントフレーム以外のコールスタック上のポインタをそれぞれ管理する．そしてこれらの表は，ともにプログラムの実行に併せて操作する．

例えば，カレントフレーム内でポインタの生成/削除が行われた場合，プロセッサはそれを検知して一次登録表を操作し，当該ポインタの登録/削除を行う．なお，一次登録表でカレントフレーム内のポインタのみを管理するためには，新たなフレームが作成された場合など，カレントフレームが遷移した際に，一次登録表の内容を遷移後のカレントフレームに対応させる必要がある．そこでこの手法では，このカレントフレームの遷移に伴う表の操作を，メソッドの開始と終了に基づいて行う．メソッドの開始時には，一次登録表に登録されているポインタを二次登録表へ退避させることで，新たに作成されるカレントフレームに対応させる．またメソッドの終了時には，一次登録表からカレントフレーム内のポインタを全て破棄し，メソッド開始時に二次登録表へ退避させておいたポインタを取得する．

以上で述べたように，プログラムの実行に併せて表を操作することで，コールスタック上に存在するポインタを全て専用表で管理できる．そして，GC 実行時には専用表を参照してポインタを取得することで，コールスタック上のポインタを判別することなく全てのオブジェクトを探索できる．つまり，従来必要とされていたポインタ判別コストを削減でき，GC の高速化が期待できる．

3.3 評価

本節では，以上で述べた手法の有効性をシミュレーションによって評価した結果について述べる．

3.3.1 評価環境

評価にはフルシステムシミュレータである gem5 シミュレータ [11] を用いた．本評価で想定するシステムの構成を**表 1** に示す．プロセッサには組み込みシステムで広く用いられる ARM アーキテクチャを選択した．ARMv7 は，32 ビットの RISC マイクロプロセッサ，ARM-RealView PBX は，ARMv7 を搭載するシステム開発用ベースボードである．そして，シミュレート実行するアプリケーション

© 2014 Information Processing Society of Japan

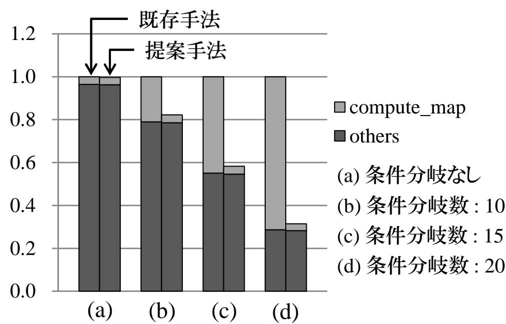


図 **2** GC 実行サイクル数

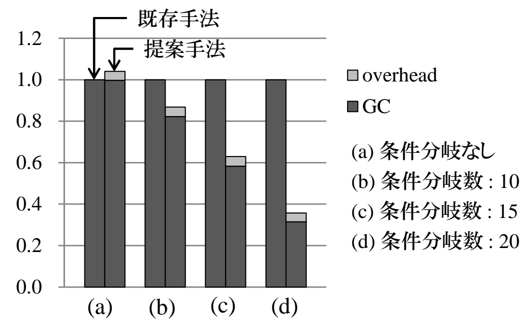


図 **3** GC 実行サイクル数に対するオーバヘッドの割合

として HotspotVM 1.6.0 を使用した．なお本評価は，オブジェクトを生成するメソッドを繰り返し呼び出すことによりメモリ領域を圧迫させるプログラムを作成し，これを HotspotVM 上で実行してサイクル数を測定した．ここで既存の HotspotVM では，ポインタ判別に要するコストを削減するために，作成したスタックマップの一部をキャッシュとして保持している．そこで，このスタックマップのキャッシュ利用効率を考慮するため，作成したプログラムには条件分岐を含まないプログラム（a）と，条件分岐を含むプログラム（b），（c），（d）を用意し，各サイクル数を比較した．なお現段階での実装は，実アプリケーションのような複雑なプログラムに対応できていないため，評価用プログラムは単純な再帰関数と条件分岐のみを用いて作成した．各プログラムはいずれも，main 関数内で再帰関数を 10 回呼び出すものである．また (a)，(b) では，関数内での再帰呼び出し回数を 10 回とした．なお (b) では，再帰関数内に条件分岐を設けることで，再帰呼び出しの際の実行フローを 10 通り作成した．また，提案手法がより有効に働くと予想されるプログラムとして，再帰回数を 15 回，条件分岐数を 15 通りに増やしたプログラム (c) と，再帰回数を 20 回，条件分岐数を 20 通りに増やしたプログラム (d) も併せて評価した．

3.3.2 評価結果

各プログラムの評価結果を図 **2** に示す．グラフは，各プログラムの GC 実行サイクル数，およびその中でスタックマップを作成する関数である compute_map() の割合を示しており，それぞれ左が既存手法，右が提案手法のサイクル数を示している．また，それぞれ既存手法の GC 実行サイクル数を 1 として正規化している．

評価結果を見ると，再帰回数，条件分岐数が増えるほど，compute_map() 関数のサイクル数が GC 全体に占める割合が増加し，GC 全体の削減率も大きくなることがわかる．これは，条件分岐による実行フローの変化にともなって，キャッシュされたスタックマップの利用効率が下がったためであると考えられる．各プログラムの削減率は，(a) の場合 0.31%の削減に止まる一方，(b)，(c)，(d) ではそれぞれ，17.8%，41.7%，68.5%の削減を確認できた．

なお，提案手法ではポインタ管理表に対する操作の際のアクセスレイテンシを，オーバヘッドとして考慮する必要がある．そこで表の操作する際のアクセスレイテンシを，表へのアクセス回数に乗じたものを提案手法におけるオーバヘッドとして概算した．その結果を図 **3** に示す．グラフを見ると，提案手法はオーバヘッドを含めた場合でも既存手法と同等，あるいはそれ以上の高速化が実現できていることがわかる．具体的には，提案手法の GC 実行サイクル数に対するオーバヘッドの比率は，(a)，(b)，(c)，(d) 各プログラムでそれぞれ 4.37%，5.58%，8.00%，13.3%となり，削減サイクル数と比較して十分に小さいことを確認した．

4. ポインタ探索の高速化

前章で述べた手法では，コールスタック上のポインタ判別に着目し，これをハードウェア支援により高速化することで，GC の実行時間を削減した．本章では，我々が提案するもう一つのハードウェア支援である，DalvikVM におけるポインタ探索処理に着目した手法 [3] について概要を述べた後，評価結果と考察を示す．

4.1 専用表を用いたクラスオブジェクトの探索

DalvikVM で管理される各オブジェクトは，クラスメソッド等のクラス情報を持つ**クラスオブジェクト**と呼ばれるオブジェクトへの参照を保持している．そのため，あるクラスに属するオブジェクトをマークした場合，その参照を辿った先のクラスオブジェクトに対してもマーク処理を施す．ここで，DalvikVM におけるマーク処理では，各オブジェクトが既にマークされているかどうかを区別しないため，同じオブジェクトへのマークが重複して行われる可能性がある．特に，上述したクラスオブジェクトは複数のオブジェクトから参照される可能性が高いため，このクラスオブジェクトへのマーク処理は，GC 処理の大きなオーバヘッドとなり得る．

そこで，このクラスオブジェクトに対するマーク処理の重複を防ぐために，マーク済みのクラスオブジェクトへの参照を管理する専用表をプロセッサに追加する．クラスオブジェクトへの参照を表で管理することで，GC 実行時に表を参照することで，各クラスオブジェクトが既にマーク

© 2014 Information Processing Society of Japan

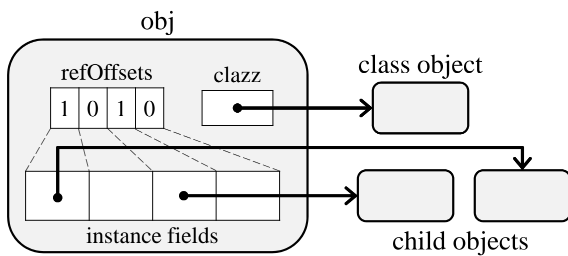


図 4　オブジェクトが持つ子オブジェクトへの参照

済みであるかどうかを判断可能となり，マーク処理の重複を防ぐことができる．

なお，この表はクラスオブジェクトを探索する際に操作する．具体的には，クラスオブジェクトを探索する前に表を確認し，そのクラスオブジェクトへの参照が既に表に登録されているかどうかを調べる．この時，参照が表に登録されていない場合には，それを表に登録し，当該クラスオブジェクトへのマーク処理を行う．一方，表に登録済みである場合，つまり当該クラスオブジェクトが既にマーク済みである場合には，これに対するマーク処理を省略し，ポインタ探索の高速化を図る．

4.2　専用表を用いた子オブジェクトの探索

前節ではクラスオブジェクトに対するマーク処理の高速化について述べた．本節では，本提案手法のもう一つの着眼点である子オブジェクトに対するマーク処理について述べる．

ここで，DalvikVM において，あるオブジェクトが持つ他のオブジェクトへの参照の例を図 4 に示す．なお図中の clazz は，前節で述べたクラスオブジェクトへの参照を保持する変数である．DalvikVM では，各オブジェクトは自身が持つ子オブジェクトへの参照を instance fields と呼ばれる配列で管理している．そしてこの instance fields の中で子オブジェクトへの参照が格納されている位置は，refOffsets と呼ばれるビットマップで管理されている．refOffsets 内の各ビットは，instance fields の各要素に対応しており，参照を保持している要素に対応するビットのみがセットされる．つまり，refOffsets 内でビットがセットされている位置に対応する instance fields の要素を取得することで，そのオブジェクトが持つ子オブジェクトへの参照を辿ることができる．

ここで，instance fields 内で子オブジェクトへの参照が格納されている位置はクラス毎に共通しているため，同一のクラスに属するオブジェクトが持つ refOffsets の値は必ず等しくなる．しかし既存の DalvikVM では，同一クラスに属するオブジェクトであっても，毎回 refOffsets の値をもとに参照を保持している instance fields の要素のオフセットを計算しており，この処理に要するコストは GC 処理のオーバヘッドとなり得る．

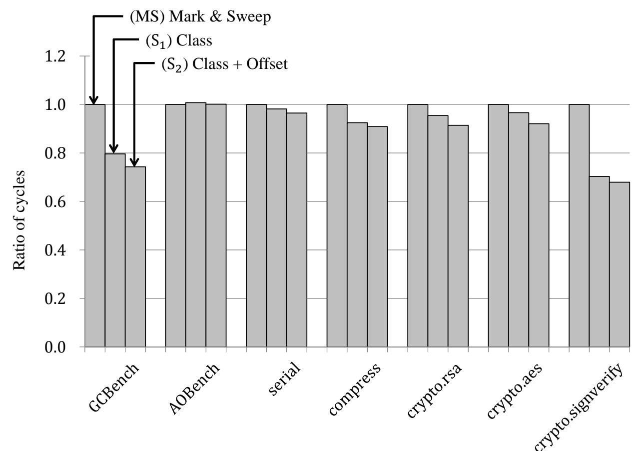


図 5　GC 実行サイクル数

そこで，前節で述べたクラスオブジェクトへの参照と併せて，各クラスの refOffsets から計算した後のオフセットの値も専用の表に記憶しておく．このオフセットの登録は，クラスオブジェクトへの参照の登録と併せて行う．この表を GC 実行時に参照することで，同一クラスのオブジェクトであれば refOffsets を用いてオフセットを計算をせずとも子オブジェクトへの参照を辿ることが可能となり，探索処理の高速化が期待できる．

4.3　評価

本節では，以上で述べた手法の有効性をシミュレーションによって評価した結果について述べる．

4.3.1　評価環境

3.3 節で用いたものと同一のシミュレータを使用し，DalvikVM を実行した．DalvikVM 上で動作させるベンチマークプログラムとしては，SPECjvm2008 からの 5 個に加え，GCBench，AOBench の計 7 個を使用した．

4.3.2　評価結果

まず図 5 に GC 全体の実行サイクル数を示す．図では各ベンチマークプログラムの結果を 3 本のグラフで示している．グラフはそれぞれ左から順に

- **(MS)**　既存の Mark & Sweep を実行するモデル
- **(S_1)**　クラスオブジェクトへの参照を表に記憶しマーク処理を省略するモデル
- **(S_2)**　(S_1) に加えて instace fileds へのオフセットを記憶するモデル

が GC の実行に要したサイクル数を示しており，(MS) の GC 実行サイクル数を 1 として正規化している．まず，既存手法である (MS) と提案手法である (S_1)，(S_2) を比較すると，AOBench のみほぼ同等の結果となっているものの，他の全てのプログラムで提案手法が GC の実行サイクル数を削減できていることが分かる．特に GCBench と crypto.signverify ではその削減率が大きく，(S_2) で最大 32.1%の GC 実行サイクル数が削減できている．この結果

© 2014 Information Processing Society of Japan

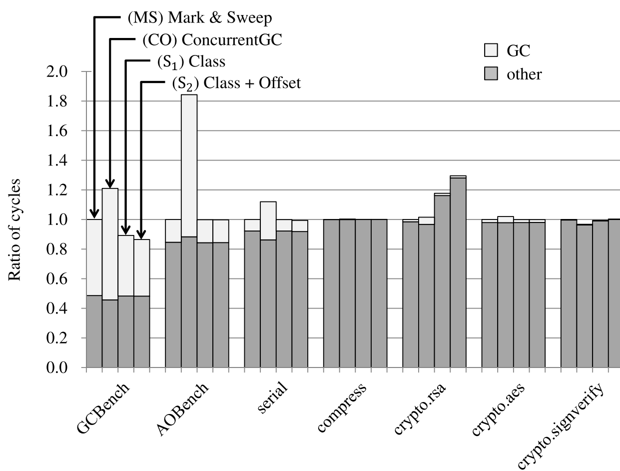


図 6　実行サイクル数

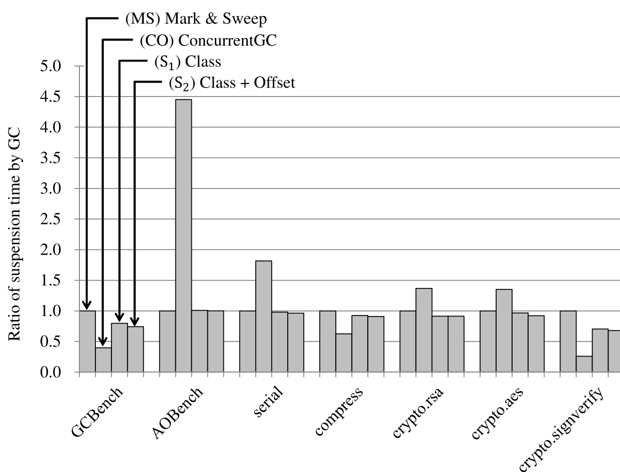


図 7　GC による平均停止時間

から，提案手法を用いることで，多くのプログラムにおいて GC の実行サイクル数を削減可能であることがわかる．

次に，ベンチマークプログラム全体の実行サイクル数と GC による平均停止時間をそれぞれ，**図 6**，**図 7** に示す．これらの図では各ベンチマークプログラムの結果を図 5 で示した 3 つのモデルに

(CO)　既存の ConcurrentGC を実行するモデル

を加えた 4 本のグラフで実行サイクル数を示しており，(MS) の実行サイクル数を 1 として正規化している．凡例は，GC の実行に要したサイクル数（‘GC’）と，GC 以外の実行に要したサイクル数（‘other’）をそれぞれ示している．

まず図 6 を見ると，GCBench，crypto.rsa 以外のベンチマークでは，あまり性能向上していないことがわかる．これは，全体に占める GC の割合が小さく，提案手法によって削減された GC の実行サイクル数が，全体の実行サイクル数に大きな影響を与えなかったためである．しかし GCBench では，全体に占める GC の割合が大きく，最大で 13.6%の高速化を実現している．なお，crypto.rsa では，(S_1) との (S_2) いずれの場合でも，GC 以外の処理である ‘other’ が増加してしまっているが，この原因に関しては現在調査中である．

次に図 7 を見ると，提案手法によって全てのベンチマークで，(MS) と比較して GC による停止時間を短縮することができている．これは提案手法によって一回の GC 実行に要するサイクル数が削減できたためである．特に，(MS) において (CO) と比較して停止時間が大きく悪化してしまっている GCBench，crypto.signverify に関しては，(CO) と比較した場合の停止時間を，(S_2) ではそれぞれ 2.5 倍から 1.9 倍，3.8 倍から 2.6 倍へと改善することができている．

以上の結果から，提案手法を用いることで，スループットを維持しつつ GC による停止時間を改善できることが確認できた．なお前章で述べた手法と同様，この手法でも，表に対する操作の際のアクセスレイテンシをオーバヘッドとして考慮する必要がある．そこで，表の操作する際のアクセスレイテンシを表へのアクセス回数に乗じたものを，提案手法におけるオーバヘッドとして概算した．その結果，提案手法の GC 実行サイクル数に対するオーバヘッドの比率は，(S_2) で平均約 1.7%となり，十分に小さいものであることが確認できた．

5.　ハードウェア支援を用いたコンパクション

本章では，現在我々が取り組んでいるハードウェア支援を用いたコンパクションについて述べる．DalvikVM では，GC アルゴリズムとして Mark & Sweep を採用している．そこで本手法では，この Mark & Sweep の各フェーズを改良し，ハードウェア支援を用いたコンパクション機能を実装する．

コンパクション機能を実現するためには，大きく分けて “オブジェクトの移動” と “移動したオブジェクトへのポインタの張り替え” という二つの処理が必要となる．次節からは，これら二つの処理の実現方法，およびそのハードウェア支援について述べ，最後に本手法を導入することで期待される効果について考察する．

5.1　ビットマップを用いた空き領域の管理

本節では，コンパクション機能に必要な “オブジェクトの移動” を実現する方法について述べる．各オブジェクトを移動するためには，まずヒープ領域内の空き領域を特定する必要がある．本手法では，これをビットマップを用いて実現する．次節からは，まず既存の DalvikVM で利用されている二つのビットマップについて述べた後，ヒープ領域内の空き領域を特定するために新たに定義するビットマップについて述べる．

5.1.1　DalvikVM におけるビットマップ

既存の DalvikVM ではヒープ領域内のオブジェクトを管理するために，Live ビットマップと Mark ビットマップと呼ばれる二つのビットマップを使用している．これらのビットマップ内の各ビットは，ヒープ領域の各アドレスと

© 2014 Information Processing Society of Japan

対応している．そして，Live ビットマップはヒープ領域内にアロケート済みのオブジェクトを，Mark ビットマップはマークフェーズにおいてマークされたオブジェクトを示すビットマップであり，それぞれ該当するオブジェクトの先頭アドレスに対応するビットがセットされる．なお，DalvikVM のメモリアラインメントは 8byte であるため，ビットマップの各ビットで 8byte 毎のアドレスを管理することで，オブジェクトが割り当てられる可能性のあるアドレスを全て管理できる．

GC 実行時には，これら二つのビットマップを利用することで，不要なオブジェクトを効率 tsu 的に回収できる．ここで不要なオブジェクトとは，ヒープ領域内にアロケートされた後，いずれのオブジェクトからも参照されなくなったオブジェクトである．つまり，Live ビットマップがセットされており，かつ Mark ビットマップがセットされていないオブジェクトのみを解放すれば良い．そのため，ヒープ全域を走査して死んでいるオブジェクトを見つける必要がなく，高速にスイープ処理を実行できる．

5.1.2　ビットマップの拡張

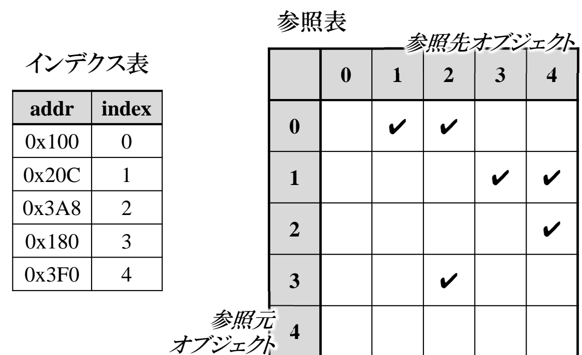
前節で述べたとおり，DalvikVM では GC 処理のために二つのビットマップを利用している．しかし，これらのビットマップはオブジェクトの先頭アドレスに対応するビットのみがセットされるため，これらを参照して空き領域を特定することはできない．そこで本手法では，オブジェクトが割り当てられている全ヒープ領域を示すビットマップを新たに定義する．以降，このビットマップを Used ビットマップと呼ぶ．

Used ビットマップは，主に Mark & Sweep におけるマークフェーズで操作する．つまり，オブジェクトに対してマークを施すと同時に，そのオブジェクトのサイズを取得し，これに応じて当該オブジェクトが割り当てられている全領域に対応するビットをセットする．これにより，オブジェクトが割り当てられている全領域をビットマップを用いて管理することが可能となり，空き領域の特定が可能となる．

なお現在の実装では，Used ビットマップは既存のビットマップと同様の方法で生成・管理している．しかし，ビットマップに対する操作では，実際のメモリアドレスとそのアドレスに対応するビットの位置を変換するための計算が必要であり，GC 処理のオーバヘッドとなり得る．そのため，今後はビットマップの管理、およびこれに対する操作をハードウェア支援によって高速化する手法についても検討していく予定である．

5.2　ハードウェア支援を用いた参照の書き換え

本節では，コンパクション機能に必要なもう一つの処理である “移動したオブジェクトへのポインタの張り替え” を実現するための方法、およびこれに対するハードウェア支援手法について述べる．その後，これを実現するために拡張した Mark & Sweep の各フェーズの動作について述べる．



インデクス表

addr	index
0x100	0
0x20C	1
0x3A8	2
0x180	3
0x3F0	4

参照表（行：参照元オブジェクト，列：参照先オブジェクト）

	0	1	2	3	4
0		✔	✔		
1				✔	✔
2					✔
3			✔		
4					

図 8　専用表の構成

5.2.1　専用表による参照関係の管理

オブジェクトの移動は，2.1 節で述べた Copying 等，いくつかの既存アルゴリズムでも必要となる処理である．ここで，同一オブジェクトに対して複数の参照が存在する場合，あるオブジェクトを移動した際には当該オブジェクトの移動先を判別できる必要がある．これは一般に，移動時に当該オブジェクトに対し forwarding pointer と呼ばれる移動先情報を付加することで対処されるが，これを順次辿るコストが比較的大きくなる．

そこで本手法では，各オブジェクト間の参照関係を記憶するための専用表をプロセッサに追加する．そしてオブジェクトの移動時には，この表を利用することで当該オブジェクトの参照元となっているオブジェクトを特定し，参照を書き換える．

この専用表の構成を**図 8** に示す．追加する専用表は大きく二つの表で構成されており，本稿ではこれらをそれぞれインデクス表，参照表と定義する．インデクス表は，オブジェクトが配置されているアドレス（addr），および各オブジェクトに固有のインデクス番号（index）を管理する．なお index の値は，各オブジェクトが表に登録される度に，それらに対して 1 ずつインクリメントされた値を付与していく．そして各オブジェクト間の参照関係は，参照表で管理する．この表はマトリックス構造とし，表の各行が参照元のオブジェクト，各列が参照先のオブジェクトにそれぞれ付与されたインデクス番号を示している．例えば図 8 に示した参照表の場合，インデクス番号が 1，つまり 0x20C 番地に格納されているオブジェクトは，インデクス番号が 3 と 4，つまり 0x180 番地と 0x3F0 番地に格納されたオブジェクトを参照していることを示している．そしてオブジェクトを移動した際には，まず移動対象のオブジェクトに付与されたインデクス番号をインデクス表から取得し，当該インデクス番号に対応する参照表の列を確認することで，ポインタを張り替えるべきオブジェクトを特定で

© 2014 Information Processing Society of Japan

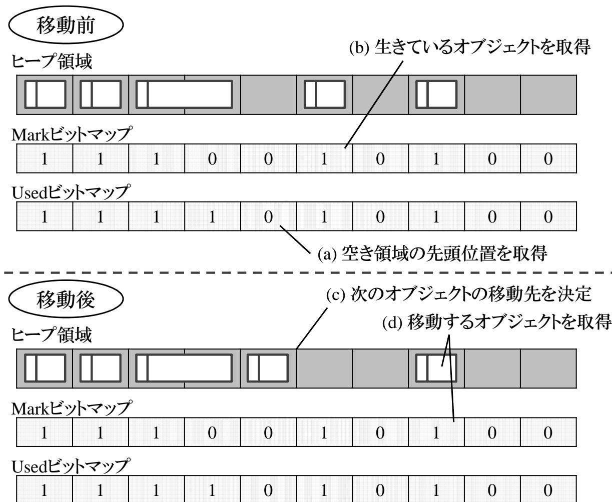


図 **9**　オブジェクトの移動

きる．

5.2.2　各フェーズの動作

提案手法では，前項で述べた専用表を用いてコンパクションを行う．これを実現するために，Mark & Sweep の各フェーズを拡張し，それぞれ専用表に対する操作を追加する．

まず提案手法における Mark フェーズでは，生きている全てのオブジェクトにマークを付けるだけでなく，各オブジェクト間の参照を辿りつつ，それらの参照関係を表に記録していく．そして表に登録された参照関係を Sweep フェーズで利用し，コンパクションを実現する．既存の Sweep フェーズでは，Mark フェーズにおいてマークの付けられなかったオブジェクトの解放処理のみが行われるが，提案する Sweep フェーズではこれにコンパクション処理を追加する．

この動作モデルを図 **9** に示す．まずヒープ領域内で使用されていない空き領域の先頭位置を，新たに定義した Used ビットマップを用いて特定する (a)．その後，Mark ビットマップを参照して，Mark フェーズにおいてマークの付けられたオブジェクト，つまり生きているオブジェクトを取得し (b)，空き領域に移動させる．この時，表に登録された参照関係をもとに，移動対象のオブジェクトを参照している各オブジェクトを全て特定し，それらの持つ参照を書き換えておく．移動が完了したら，当該オブジェクトのサイズをもとに，そのオブジェクトの末尾を次のオブジェクトを移動させる先頭位置として決定する (c)．そして再び Mark ビットマップを参照し，ビットがセットされているオブジェクト，つまり次に移動するオブジェクトを取得する (d)．この一連の動作を，マークビットマップ内でビットがセットされている全てのオブジェクトに対して行うことで，ヒープ領域のコンパクションが実現できる．

5.3　期待される効果

GC 実行時にコンパクションを行うことで，フラグメンテーションの発生を防ぎ，メモリ利用効率の向上が期待できる．これにより，アロケーションの失敗に起因する GC の発生を抑制し，GC の実行回数自体の削減も可能になると考えられる．さらに，各オブジェクトが比較的近い領域に格納されることになるため，空間的局所性によりキャッシュの利用効率も向上しやすくなる．また，コンパクションによって空き領域が一つの連続したメモリ領域として存在するようになるため，アロケーションが高速に行えることも期待できる．このようなキャッシュ利用効率の向上やアロケーションの高速化は，アプリケーションのスループット向上に繋がるものである．つまり，本手法は GC の性能改善のみならず，システム全体の性能向上にも繋がることが期待できる．

また今後のさらなる改良として，移動先をより柔軟に決定することが考えられる．一般に，生成されるオブジェクトの寿命に関しては，“オブジェクトの多くは生成して間もなく不要になる” という経験則が知られている．そこで，寿命が長いと予想されるオブジェクトを優先的にヒープ領域の先頭から詰めていくことで，解放される領域をある程度限定でき，よりメモリの利用効率向上に繋がると期待できる．

6.　まとめと今後の課題

これまで，我々はハードウェア支援によって GC を高速化させる手法を提案してきた．本稿では，我々が提案している二つの手法を紹介し，シミュレーションによる評価結果，および手法の有用性を示した．また，現在取り組んでいるコンパクション機能を備えたハードウェア支援型 GC について概説し，手法の導入によって期待される効果について述べた．

今後の課題は，実際にコンパクション機能を実装し，その有用性を確認することである．また，表のサイズやアクセスレイテンシなどのハードウェアコストだけでなく，ハードウェアの拡張に伴う消費エネルギーの増加量についても調査し，これらを抑制できるような手法を模索していきたい．

謝辞　本研究の一部は，JSPS 科研費 25540019，および稲盛財団研究助成金による．

参考文献

[1] Ossia, Y., Ben-Yitzhak, O., Goft, I., Kolodner, E. K., Leikehman, V. and Owshanko, A.: A Parallel, Incremental and Concurrent GC for Servers, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'02)*, pp. 129–140 (2002).

[2] Ideue, K., Satomi, Y., Tsumura, T. and Matsuo, H.: Hardware-Supported Pointer Detection for common

© 2014 Information Processing Society of Japan

Garbage Collections, *Proc. 1st Int'l Symp. on Computing and Networking (CANDAR'13)*, pp. 134–140 (2013).

[3] 里見優樹，井手上慶，津邑公暁，松尾啓志：GC におけるポインタ探索高速化のためのハードウェア支援手法，情処研報，Vol. 2013-ARC-207, No. 27, pp. 1–9 (2013).

[4] McCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, *Communications of the ACM*, Vol. 3, pp. 184–195 (1960).

[5] Minsky, M.: A LISP Garbage Collector Algorithm Using Serial Secondary Storage, Technical report, Massachusetts Institute of Technology (1963).

[6] Collins, G. E.: A Method for Overlapping and Erasure of Lists, *Communications of the ACM*, Vol. 3, pp. 655–657 (1960).

[7] 中村成洋，相川　光，竹内郁雄：ガベージコレクションのアルゴリズムと実装，秀和システム (2010).

[8] Takeuchi, I., Yamazaki, K., Amagai, Y. and Yoshida, M.: Lisp can be "Hard" Real Time, *Proc. Japan Lisp User Group Meeting (JLUGM)* (2000).

[9] Click, C., Tene, G. and Wolf, M.: The Pauseless GC Algorithm, *Proc. 1st ACM/USENIX Int'l Conf. on Virtual Execution Environments (VEE'05)*, pp. 46–56 (2005).

[10] Bak, L., Duimovich, J., Fang, J., Meyer, S. and Ungar, D.: The New Crop of Java Virtual Machines, *Proc. 13th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pp. 179–182 (1998).

[11] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D. and Wood, D. A.: The gem5 Simulator, *ACM SIGARCH Computer Architecture News*, Vol. 39, pp. 1–7 (2011).

© 2014 Information Processing Society of Japan