Vol.2013-ARC-207 No.27
Vol.2013-HPC-142 No.27
2013/12/17

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

里見 優樹[1]　井手上 慶[1]　津邑 公暁[1]　松尾 啓志[1]

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 の高速化はこれまで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，GC が抱える重要な問題の根本的解決には未だ至っていない．そこで本稿では，多くの GC アルゴリズムにおいてオブジェクト間のポインタを辿ってメモリ上のオブジェクトを探索する必要がある点に着目し，これをハードウェア支援によって高速化する手法を提案する．オブジェクトの多くはクラスメソッド等の情報を持つクラスオブジェクトを参照している．そのため，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く，クラスオブジェクトへのマーク処理の多くが重複してしまっている．そこで，一度探索したクラスオブジェクトへのポインタを管理する専用の表を用いることで，クラスオブジェクトへのマーク処理の重複を防ぐ。また，オブジェクトへの参照を辿る際には，その参照が参照元オブジェクト内のどこに存在するかを計算によって求める必要があるが，この参照が存在する位置はオブジェクトのクラスごとに共通している．そこで，クラスオブジェクトの値と併せて，子オブジェクトへの参照の位置を表で管理することで，オブジェクト間の参照の探索コストを削減し GC の高速化を実現する．シミュレーションによる評価の結果，提案手法による GC の高速化が，スループット及び GC による停止時間を改善させることを確認した．

1. はじめに

スマートフォンなど，一般にメモリ容量が少なく，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 (GC)** の性能が与える影響範囲が拡大している．一方，GC の高速化はこれまで主にアルゴリズムの改良という観点から研究されてきているにも関わらず，GC 実行時のプロセス全停止によるレスポンス低下など，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．そこで本稿では，多くの GC アルゴリズムで，オブジェクト間のポインタを辿ってメモリ上のオブジェクトを探索する必要がある点に着目し，これをハードウェア支援によって高速化する手法を提案する．

2. ガベージ・コレクション

本章では，まずガベージ・コレクションについて概説する．その後，ハードウェア支援による GC 高速化と，その既存研究について述べる．

[1] 名古屋工業大学
Nagoya Institute of Technology

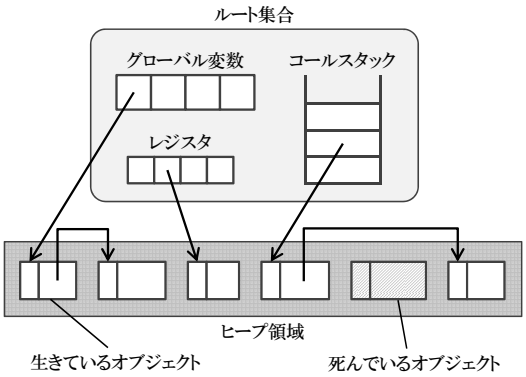

図 1　プログラム実行時のヒープ領域と参照関係の様子

2.1 ガベージ・コレクションによるメモリ管理

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** [1] 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．なお，Mark & Sweep では参照を辿りながらオブジェクトを探索するため，参照関係を書き換えないよう GC の実行中はアプリケーションの動作を停止させなければならない．これが GC 実行時のレスポンス低下の原因となっている．

また，その他の代表的な GC アルゴリズムとして，Copying [2] と Reference Counting [3] 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 [4]．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 ハードウェア支援による GC の高速化

はじめに述べたように，GC には実行時のプロセス全停止によるレスポンス低下などの問題が存在する．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システムの停止時間を短縮する ConcurrentGC [5] などのアルゴリズムが提案されている．しかし，例えば ConcurrentGC では並行動作のためのオーバヘッドによってスループットが犠牲になるなど，GC の抱える問題の根本的な解決には至っていない．

従来，システムの利用目的や特徴を考慮してアルゴリズムやパラメータをチューニングすることで，GC 実行によるプロセス停止やスループットの悪化などの問題に対処してきたが，その作業はプログラマの大きな負担となっている．そこで本稿では代表的 GC アルゴリズムに共通して必要となる処理要素に着目し，これを高速化するハードウェア支援手法を提案する．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実現するだけでなく，GC の高速化を，チューニングに頼らないソフト・ハードウェアの協調問題として発展させることを目指す．

2.3 関連研究

ハードウェアによる GC 支援には，わずかながら既存研究が存在する．それが SILENT [6] と Network Attached Processing(NAP) [7] である．

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時，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 社の開発した，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，アプリケーションがポインタが参照しているオブジェクトをロードする度，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

Vol.2013-ARC-207 No.27
Vol.2013-HPC-142 No.27
2013/12/17

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．これに対して本研究は代表的 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点でこれらの既存手法と異なる．なお，これまでに我々は同様の着眼点で，コールスタック上のポインタを専用の表で管理することで，GC 実行時のポインタ判別処理を高速化するハードウェア支援手法を提案している [8]．この手法は，コールスタックに含まれるポインタと即値の判別にコストがかかっていることに着目したものである．コールスタックにポインタを格納する際に，同時にそのポインタを専用の表に登録しておき，GC 実行時にその表をルート集合の一つとして参照することでコールスタックの走査を省略し GC 実行を高速化する．しかしながら，このポインタの判別処理は「正確な GC」と呼ばれる，ポインタと即値を厳密に区別する実行形式でのみ必要な処理である．だがモバイル機器など，スループットが重要視される環境では「保守的 GC」と呼ばれる，厳密にポインタと即値を区別しない実行形式が用いられることがある．例えば，モバイル機器の実行環境として代表的な Android の DalvikVM でも保守的 GC が採用されている．こうした場合にはこの手法を適用することはできない．これに対し本稿では，オブジェクト間の参照の探索という代表的 GC アルゴリズムで共通して必要となる処理に着目する．これにより，GC が与える影響をモバイル機器などを含めたより広い範囲で改善させることを目指す．そこで，本稿では先に述べた Android の実行環境である DalvikVM を対象としてハードウェアによる GC の高速化手法を提案する．

3. DalvikVM における GC の概要

本章では，本研究で対象とする DalvikVM に実装されている GC について述べる．

3.1 ConcurrentGC

DalvikVM では GC アルゴリズムに Mark & Sweep をベースとした ConcurrentGC を採用している．これは，モバイル機器ではユーザ体感品質の観点から GC による停止時間が短いことが求められるためである．

しかし，2.2 節で述べたように，ConcurrentGC ではアプリケーションと GC の並行動作によってスループットが犠牲となってしまう．**図 2** と**図 3** に示すグラフは，通常の Mark & Sweep と，アプリケーションと Mark & Sweep が並行動作する ConcurrentGC の性能を比較したものである．それぞれ，図 2 は GCBench，AOBench，および SPECjvm2008 から 5 個の計 7 個のベンチマークプログラムを実行した際の GC による平均停止時間，また図 3 は

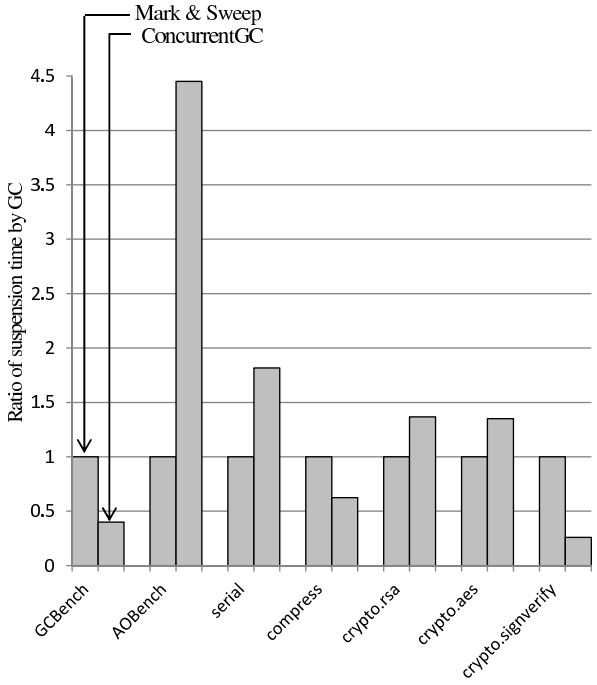

図 2 GC による平均停止時間

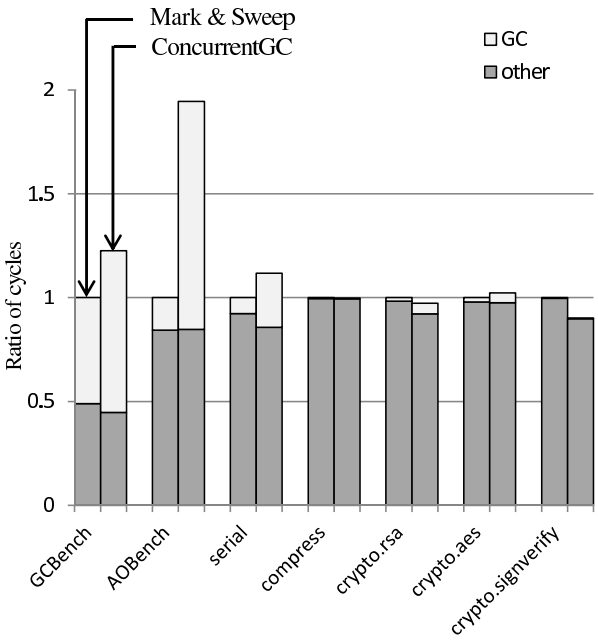

図 3 ConcurrentGC によるスループットの悪化

同じベンチマークの総実行時間とそれに占める GC の割合を示しており，いずれのグラフも Mark & Sweep の結果を 1 として正規化している．まず，図 2 を見ると GCBench，compress，signverify では ConcurrentGC によって GC による停止時間が短く抑えられていることが分かる．しかし，それ以外のプログラムでは逆に ConcurrentGC の方が停止時間が長くなってしまっている．これは，これらのプログラムは一回の GC 実行に要する時間が短く，スレッド同期などの並行動作のためのオーバーヘッドの割合が相

Vol.2013-ARC-207 No.27
Vol.2013-HPC-142 No.27
2013/12/17

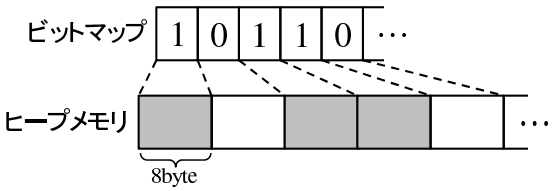

図 4　ビットマップによるオブジェクト管理

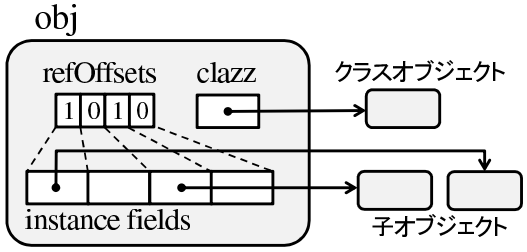

図 5　オブジェクトが持つ子オブジェクトへの参照

対的に大きくなってしまったためだと考えられる．また，図 3 に示すように，並行動作のためのオーバヘッドによって，実行時間に占める GC の割合は全てのプログラムで増加している．特に GCBench， AOBench， serial では GC の実行時間増加によってスループットが大きく悪化してしまっている．これらの結果より，一般的に ConcurrentGC は，スループットを犠牲にして GC による停止時間を短縮させる目的で使用されるが，場合によってはスループットだけでなく停止時間も悪化してしうことが分かる．特に，モバイル機器におけるスループットの低下はバッテリー持続時間の悪化に繋がってしまう．そのため GC 性能の向上を考える場合，単に停止時間を短縮するだけでなく同時に GC の動作を高速化しスループットの向上を図ることも重要となる．そこで，本稿では Mark & Sweep をベースとして，ハードウェアで GC を高速化することでスループットと停止時間の両方の改善を目指す．

3.2　オブジェクトを参照するポインタの探索

前節で述べたように DalvikVM の GC アルゴリズムは Mark & Sweep をベースとしている．単純な Mark & Sweep ではオブジェクトのヘッダ内にマーク用のビットを確保するのが一般的で，オブジェクトへマークする際はオブジェクトへの書き込みが発生することになる．しかし，これはコピーオンライトとの相性が悪く，特にメモリ容量が限られるモバイル機器ではメモリコピーの頻発によってメモリ領域が圧迫されてしまう．

そこで DalvikVM では，**ビットマップマーキング**という手法でこの問題に対処している．ビットマップマーキングとは，ビットマップと呼ばれるビット列でオブジェクトへのマークを管理する手法である．ビットマップマーキングで使用されるビットマップは，各オブジェクトに対応するマークビットをオブジェクト外の領域に集めたものである．ビットマップを用いて，メモリ上でオブジェクトが存在する位置を表した様子を**図 4** に示す．DalvikVM のメモリアライメントは 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することができる．ビットマップマーキングでは，オブジェクトとは別の領域でマークを管理することで，オブジェクトが割り当てられているメモリ領域の書き換えを防ぐことができ，マーク操作によるメモリ領域のコピーを抑制することができる．

ここで DalvikVM における，あるオブジェクト (obj) が持つ子オブジェクトへの参照の例を**図 5** に示す．obj 内で，マークが必要なオブジェクトへの参照は，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への参照 (clazz)，及び子オブジェクトへの参照を格納する配列 (instance fields) に存在する．GC 実行時にオブジェクトをマークする際は，まず最初にクラスオブジェクトへのマークを行う．そこで clazz に格納されている値から，ビットマップ中のどのビットがクラスオブジェクトのメモリアドレスに対応するかを計算し，そのビットをセットする．次に instance fields から参照されているオブジェクトをマークする．instance fields 中で，オブジェクトへの参照が格納されている位置は refOffsets というビットマップで示されている．この refOffsets は，その 1 ビットが instance fields の一要素と対応しており，refOffsets 内のセットされているビットに対応する instance fields の要素に子オブジェクトへの参照が格納されている．そこで，instance fields から参照されているオブジェクトをマークする時は，refOffsets から instance fileds で参照が格納されている場所へのオフセットを計算してアクセスする必要がある．オフセットの値を計算するにはまず，そのビットがビットマップの先頭から何ビット目なのかを求める．そして，そのビット数に instance fields の一要素分のサイズを乗じたものが，参照が格納されている場所へのオフセットとなる．

このようにして，オブジェクトが持つ子オブジェクトへの参照を辿り，新たに発見したオブジェクトについて更なる子オブジェクトを探索する．こうした動作を再帰的に繰り返すことで，メモリ上のすべての生きているオブジェクトにマークできる．

しかし，こうしたビットマップマーキングや refOffsets を使用した参照の探索など，ビットマップを用いた処理では，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，例えばビットマップマーキングでは，オブジェクトのヘッダへのマーキングと比較してマーク処理が遅くなってしまうといった問題が存在する．更に，DalvikVM では既にマーク済みの

Vol.2013-ARC-207 No.27
Vol.2013-HPC-142 No.27
2013/12/17

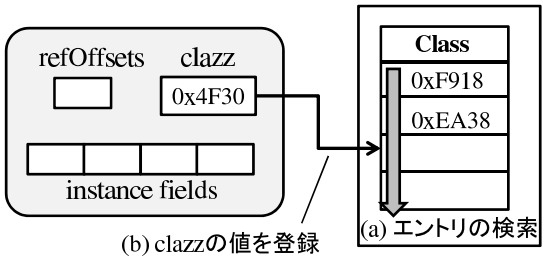

図 6　クラスオブジェクトへの参照の登録

```
1 while(refOffsets != 0){
2   rshift  = count_leading_zero(refOffsets );
3                                  // 連続したゼロをカウント
4   offset  = calc_offset (rshift ); // オフセットの計算
5   ref  = get_pointer(offset );     // ポインタの取得
6   mark_object(ref);              // オブジェクトにマーク
7   clear_bit (rshift );             // ビットを降ろす
8 }
```

図 7　instance fileds の探索

オブジェクトへの参照でもそのオブジェクトがマーク済みかどうか区別せずにマーク処理を行うため，同じオブジェクトへのマークが重複して行われる場合が存在する．特にクラス情報を持つクラスオブジェクトは複数のオブジェクトから繰り返し参照されることが多く，無駄なマーキングによって一層マーク処理のコストが大きくなっている．

4. ハードウェア支援によるポインタ探索高速化

本章では，ハードウェア支援によってマーク処理時のポインタ探索を高速化する手法を提案する．提案手法ではマーク済みのクラスオブジェクトや instance fields のオフセットを専用の表で記憶しておくことで，同一クラスのオブジェクトが持つポインタの探索に要するコストを削減し GC 実行を高速化する．

4.1　クラス情報を用いたマーク処理の省略

3.2 節で述べたように，オブジェクトは自身のクラスオブジェクトへの参照を持ち，これを毎回マークしている．しかし一般に，プログラム中のクラスの種類は限られており，同一のクラスオブジェクトに対するマーク処理が重複して行われてしまう可能性が高い．そこで，クラスオブジェクトへの参照を表に記憶しておき，マーク時にこれを参照することで，マーク処理の重複を防ぐ手法を提案する．

この手法の動作モデルを**図 6** に示す．追加する表はクラスオブジェクトへのポインタの値を格納するフィールド (Class) を持つ．なおこの表は，高速な連想検索が可能な汎用 3 値 CAM (Content Addressable Memory) での実装を想定しており，参照の値が表に登録されているかどうかを高速に検索可能である．

クラスオブジェクトをマークする際は，まずクラスオブジェクトへの参照の値で表を検索する (図 6(a))．参照の値が登録されていなかった場合は，その値を表へ登録する (図 6(b))．登録の際，表に空きがない場合，値の登録は行わない．なお，もし仮に表が溢れた場合でも，参照が表に登録されていない時は，通常通りビットマップへのマーク処理を行うため，GC の実行に支障を来すことはない．

次に，マークしようとするクラスオブジェクトが表に登録されていた場合，エントリの検索がヒットするため，そのクラスオブジェクトへのマーク処理を省略する．現在の実装では，ビットマップへのマークを行う関数の呼び出し時に，プログラムカウンタをインクリメントしてその関数の呼び出しをスキップするようにしている．こうすることで，同一のクラスオブジェクトへのマーク処理を省略することが可能となる．

このようにクラスオブジェクトへの参照を表で管理することで，マーク処理の前の段階でそのオブジェクトがマーク済みかどうかを判別することが可能となる．これによって，同一クラスオブジェクトへの重複したマーク処理そのものが省略可能となり，3.2 節で述べたポインタ探索を高速化できると考えられる．

4.2　子オブジェクトへのポインタ計算の高速化

前節では同一クラスのオブジェクトが同じクラスオブジェクトへのポインタを持つことに着目し，そのポインタの値を表に記憶しておくことでクラスオブジェクトへの無駄なマーキングを省略する手法を提案した．しかし，同一クラスのオブジェクトに共通する情報は他にも存在する．それが，instance fields 内のどこに子オブジェクトの参照があるかを表す，refOffsets の値である．3.2 節で述べたように，refOffsets はビットマップである．そのため，instace fields から子オブジェクトへのポインタを取得するためにオフセットを計算をする必要があるが，この処理はオブジェクトを探索する度に実行され，大きなコストとなっている．しかし，同一クラスのオブジェクトは同じ子オブジェクトを持つため，refOffsets の値も同じになる．そこで，クラスオブジェクトへのポインタと併せて refOffsets から計算した後のオフセットの値も表に記憶しておく手法を提案する．

オフセットの値の登録は，クラスオブジェクトへの参照の登録時，つまりそのクラスのオブジェクトを初めて探索する時に行う．3.2 節で述べたように，DalvikVM ではクラスオブジェクトへのマークを行った後に，instance fields から参照されているオブジェクトをマークする．この時の動作を簡易的に表したコードを**図 7** に，またその時の refOffsets

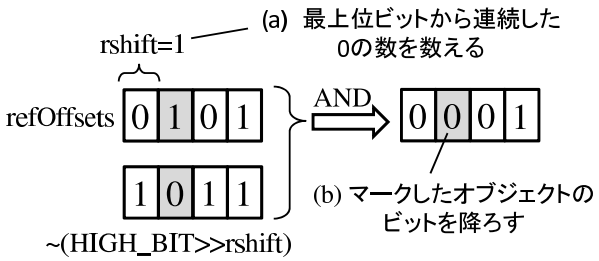

図 8　instance fields 探索時の refOffsets への操作

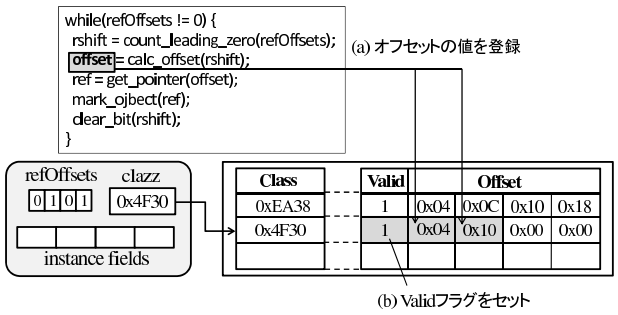

図 9　オフセットの値を表へ登録

に対する操作の様子を図 8 に示す．count_leading_zero() は refOffsets の最上位ビットから連続した 0 の数を数える関数である．この値 (rshift) によって，セットされているビットで最も上位のものが先頭から何ビット目かが分かる (図 8(a))．calc_offset() は rshift の値から子オブジェクトへの参照が格納されている位置を計算する関数である．その後，get_pointer() で格納されている参照の値を取得し，mark_object() でそのオブジェクトへのマークを行う．そして最後に clear_bit() で，今マークしたオブジェクトに対応する refOffsets のビットを降ろす．そのために，最上位ビットのみがセットされたビット列 (HIGT_BIT) を rshift 分右シフトし，その否定をとる．こうすることでマークしたオブジェクトに対応するビット位置のみが 0 のビット列が得られる．これで refOffsets をマスクすることで，マークが完了したオブジェクトのビットのみを降ろす (図 8(b))．こうすることで，refOffsets 中のセットされている次のビット位置を count_leading_zero() によって求めることができる．この動作を refOffsets が 0 になるまで繰り返すことで，instance fileds 内の全ての参照を辿ってオブジェクトにマークすることが可能となる．この動作の中で，図 7 の 4 行目で得られた offset が，記憶するべきオフセットの値である．

そこで，この値を記憶しておくための表を追加する．この表はオフセットを格納するフィールド (Offset) と，そのエントリに有効なオフセットが登録されているかを示すフラグ (Valid) を持つ．この表と前節で追加した表は同数のエントリを持ち，各エントリが一対一に対応する．また Offset へのアクセスは，探索対象オブジェクトが持つクラスオブジェクトへの参照の値による連想検索がヒットしたエントリのインデックスを用いて行う．そのため，Offset と Valid で構成される表はインデックスによるアクセスが可能な RAM による実装を想定している．

追加した表に，instance fileds へのオフセットを登録する際の動作の様子の例を図 9 に示す．オフセットの値の登録は図 7 で示した動作に併せて行う．calc_offset() で offset を計算するのと同時に，その値を表へ登録する (図 9(a))．refOffsets から得られる全てのオフセットの値を表に登録し終わったら，Valid フラグの値を 1 にセットする (図 9(b))．ただし，refOffsets の値が最初から 0，すなわち子オブジェクトを持たない場合や，表にオフセットが登録しきれない場合はオフセットの登録を行わず，Valid フラグもセットしない．これによって，このエントリに利用可能なオフセットが登録されているかどうかを，Valid フラグを確認することで判別することができる．

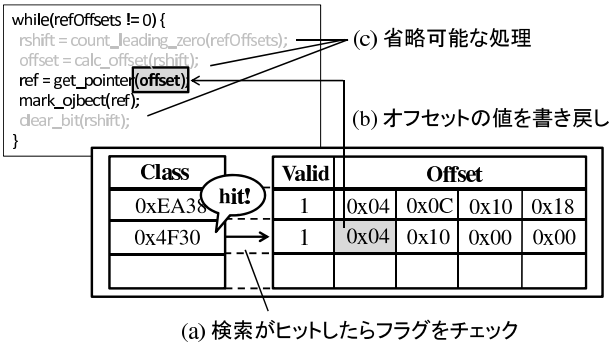

図 10　表を用いたオフセット計算の省略

この後，同一クラスのオブジェクトの instance fileds を探索する場合は，登録したオフセットの値を利用して子オブジェクトへのポインタ取得を高速化する．この時の動作の様子を図 10 に示す．探索するオブジェクトが表に登録済みの場合，クラスオブジェクトの値による検索がヒットする．この時，ヒットしたエントリに対応する Valid フラグをチェックし，値が 1 だった場合は，オフセットの値が利用可能であると判断する (図 10(a))．そして，表からオフセットの値を get_pointer() への引数として対応するレジスタへ書き戻す (図 10(b))．これを，表に登録されているオフセットの分だけ繰り返す．このように instance fileds へのオフセットの値を表に登録しておき，同一クラスのオブジェクトを探索する際にその値を利用することで，オフセットの計算に必要な処理が省略でき (図 10(c))，子オブジェクトへのポインタの探索を高速化できると考えられる．なお，仮にオフセットが登録しきれなかった場合でも，Valid フラグが 0 の場合は通常の子オブジェクトの探索を行うため，プログラムの実行には支障を来さない．

5. 評価

本章では，提案手法の有効性をシミュレーションにより

Vol.2013-ARC-207 No.27
Vol.2013-HPC-142 No.27
2013/12/17

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

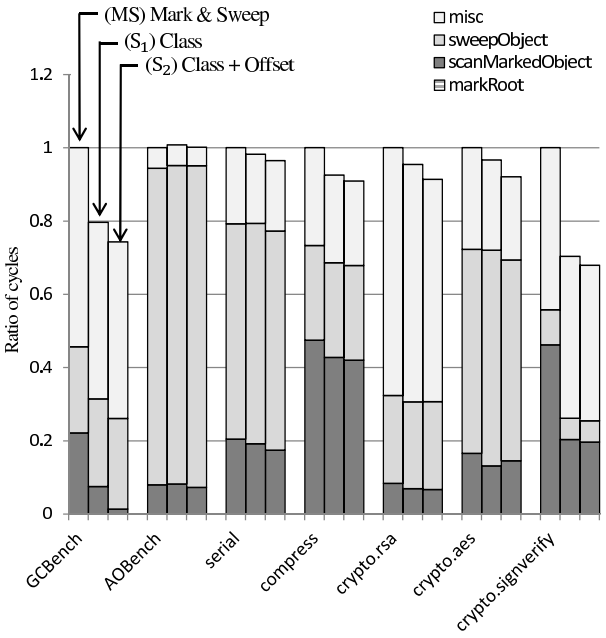

図 11　GC 実行サイクル数

評価し，得られた評価結果から考察を行う．

5.1　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 [9] 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．

DalvikVM 上で動作させるベンチマークプログラムとしては GCBench，AOBench，SPECjvm2008 から 5 個の計 7 個を使用した．

5.2　評価結果

まず**図 11** 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．3 本の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．また，凡例は内訳を示しており，markRoot はルート集合からのマーク，scanMarkedObject は子オブジェクトのマーク，sweepObject は不要なオブジェクトの回収，misc はスレッド同期など Mark & Sweep 以外の処理に要したサイクル数をそれぞれ示している．

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 を除く全てのプログラムで (S_1)，(S_2) が (MS) と比較して GC の実行サイクル数を削減できている．これは，提案手法によって scanMarkedObject が削減されたためである．また AOBench に関しても (MS) とほぼ同等の結果となっている．特に，GCBench と crypto.signverify の 2 つのプログラムに関しては，(MS) において scanMarkedObject が占める割合が大きく，提案手法が特に有効に働いたため，(S_2) で最大 32.1%の GC 実行サイクル数が削減できている．一方で, serial と compress では (MS) に占める scanMarkedObject が先の 2 つのプログラムと同程度にもかかわらず，提案手法の効果はあまり大きくない．これは GCBench や crypto.signverify では限られた種類のオブジェクトが多く生成されるのに対して，serial や compress では生成されるオブジェクトの種類が多く，表に登録した情報が有効に使用されなかったためである．

また，(S_1) と (S_2) を比較すると，全てのプログラムで (S_2) の方が性能向上している．しかし，例えば GCBench, crypto.rsa，crypto.aes では約 5%高速化しているのに対して AOBench では約 0.7%の高速化に留まるなど，その効果には幅がある．この理由として，性能向上幅が大きいプログラムでは複数の子オブジェクトを参照するオブジェクトが多く存在するのに対して，性能向上幅が小さいプログラムではそういったオブジェクトはあまり存在せず，オフセットの登録があまり行われなかったため，提案手法が有効に働かなかったことが挙げられる．

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 12**，**図 13** 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 11 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．

まず図 12 を見ると，GCBench の (S_2) で最大 13.6%の高速化を実現している．これは，GCBench では全体の実行サイクル数に占める GC の割合が大きく，提案手法によって GC の実行サイクル数が削減されたことが大きく影響している．一方，crypto.rsa を除くその他のベンチマークプログラムでは 2 つの提案手法のいずれも実行サイクル数は (MS) と同程度となっている．これは GCBench の場

Vol.2013-ARC-207 No.27
Vol.2013-HPC-142 No.27
2013/12/17

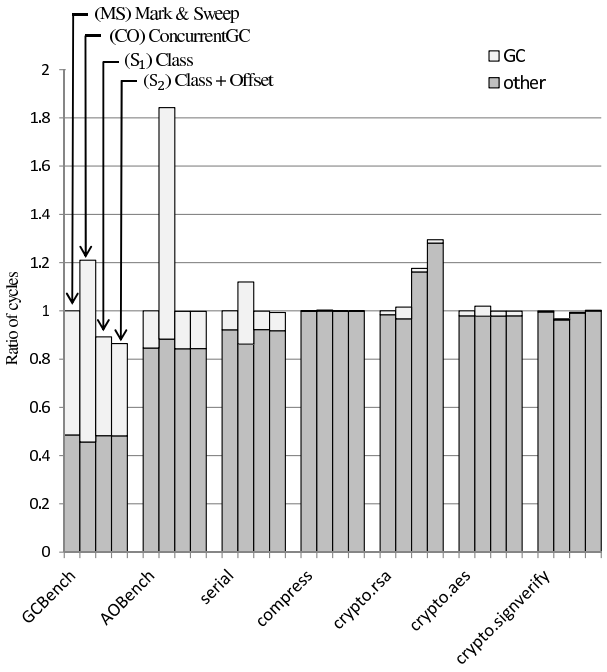

図 12　実行サイクル数

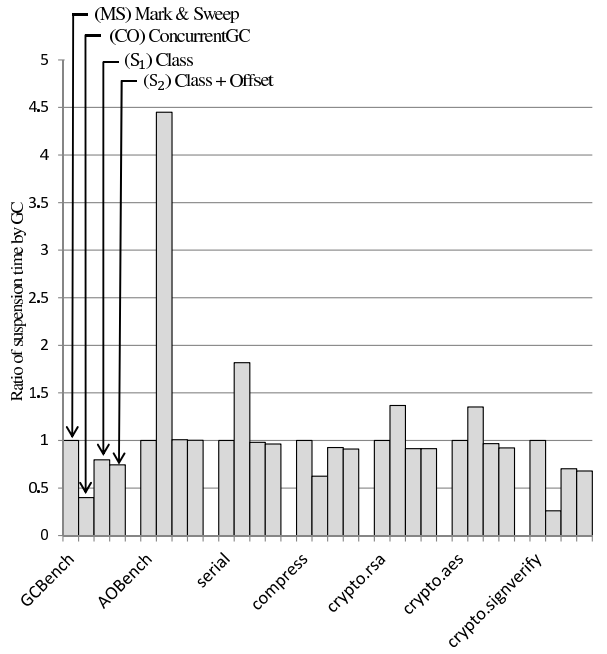

図 13　GC による平均停止時間

合とは逆に全体の実行サイクル数に占める GC の割合が小さく，GC の高速化がスループットの改善にほとんど影響しないためである．なお crypto.rsa では，(S1) と (S2) のいずれの場合でも，GC 以外の処理である other の実行サイクル数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．また図 13 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，(S2) 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．

これらの結果をまとめると，提案手法によって GC を高速化することで，スループットを維持しつつも，GC による停止時間を改善させることができた．なお，提案手法では GC 実行サイクル数を削減できる一方，表に対する操作の際のアクセスレイテンシをオーバーヘッドとして考慮する必要がある．本評価では，提案手法で追加する表で使用する RAM は，L1 キャッシュと同様のものを想定している．そこで RAM に対するアクセスレイテンシを，シミュレート対象のマシンにおける L1 キャッシュへのアクセスレイテンシと等しい 2 サイクルと仮定する．また，使用する CAM は，MOSAID 社の DC18288 [10] を参考にし，その動作周波数を 200MHz と想定する．CAM の連想検索は 1 サイクルで可能であるが，シミュレート対象の CPU の動作周波数は 2GHz であるため，周波数比を考慮して CPU サイクルに換算すると 10 サイクルと考えることができる．

以上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，(S2) 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．

また，ベンチマークプログラムの実行結果から，クラスオブジェクトの種類はおよそ 200 以下であることや，オブジェクトが参照する子オブジェクトの数はほぼ 4 つ以下であるということが分かっている．そのため提案手法で追加する表のサイズは 200 エントリ，Offset の登録数は 4 を想定している．従って，クラスオブジェクトの参照を登録する表は幅 32bit 深さ 200 行の CAM で構成可能であると考えられる．また同様に，オフセットの値を登録する表に関しても，ベンチマークの実行結果では，オフセットの値は高々 32 であったことから，Offset は 5bit * 4 = 20bit に Valid フラグの 1bit を加えた，幅 21bit 深さ 200 行の RAM で構成可能であると考えられる．そのため，提案手法で必要となる追加ハードウェアは，二つの表を使用する (S2) の場合でも，合計 1325byte と少量である．なお，仮に表が溢れた場合でも，4.1，4.2 節で述べた通り，表を使用しない通常の処理を実行するため GC の実行に支障を来すことはない．

6. おわりに

本稿では，多くの GC アルゴリズムの高速化を目的として，代表的 GC アルゴリズムに共通して必要となる，オブジェクト間の参照探索に着目し，これを高速化するハードウェア支援手法を提案した．シミュレーションによる評価

Vol.2013-ARC-207 No.27
Vol.2013-HPC-142 No.27
2013/12/17

の結果，提案手法による GC の高速化が，スループット及び GC による停止時間を改善させることが確認できた．

本研究の今後の課題として，提案手法によるハードウェアを拡張に伴って増加する消費エネルギーの調査が挙げられる．今後は，消費エネルギーを抑制して低消費電力を実現できるような手法も模索していく．

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

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

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

[4] 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).

[5] 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).

[6] Takeuchi, I., Yamazaki, K., Amagai, Y. and Yoshida, M.: Lisp can be "Hard" Real Time.

[7] 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).

[8] Ideue, K., Satomi, Y., Tsumura, T. and Matsuo, H.: Hardware-Supported Pointer Detection for common Garbage Collections, *Proc. 1st Int'l Symp. on Computing and Networking (CANDAR'13)*, to appear (2013).

[9] 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).

[10] MOSAID Technologies Inc.: *Feature Sheet: MOSAID Class-IC DC18288*, 1.3 edition (2003).