Vol.2016-ARC-221 No.18
2016/8/9

GCにおける冗長なマーク処理抑制のためのハードウェア支援手法

河村 慎二[1]　津邑 公暁[1]

概要：モバイル機器の普及に伴い，ガベージ・コレクション（GC）の性能が及ぼす影響範囲が拡大している．一方，GC は古くからプロセス全停止に伴うレスポンス低下が問題視されており，これを解決するために主にアルゴリズム面で改良がなされてきた．しかし，それらではスループットが犠牲になるなど，GC が抱える根本的な問題解決には至っていない．これに対し我々は，GC をハードウェア支援することでこれを解決することを目指している．本稿では，モバイル端末の実行環境として広く普及している DalvikVM に実装されている GC の処理において，同一オブジェクトに複数回マークを施している点に着目し，これを高速化するハードウェア支援手法を提案する．本手法では，マーク済みのオブジェクトを管理するための専用表をプロセッサに追加し，オブジェクトの探索時にこの専用表を参照することで，マーク済みのオブジェクトに対する冗長なマーク処理を省略する．シミュレーションによる評価の結果，提案手法により，最大 12.4%，平均 5.7%の性能向上を確認した．

1. はじめに

スマートフォンなど，小容量のメモリしか搭載されておらず，メモリ管理機能の重要性が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．この GC に関しては古くから，サーバサイド Java 環境など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．このような問題を解決するため，これまで主にアルゴリズムの改良という観点から多くの研究がなされてきた．しかし，それらはシステムの構成や実行するアプリケーションに合わせた煩雑なチューニングによって GC の発生頻度を抑えるものや，スループットを犠牲にしてシステムのレスポンスを改善するものがほとんどであり，GC が抱える問題の根本的な解決策とは成り得ていない．

そこで我々は，多くの実行環境で用いられる代表的な GC アルゴリズムに共通して存在する構成処理要素に着目し，これをハードウェア支援によって高速化することを目指している．本研究にあたり，まずモバイル端末の実行環境として広く普及している **DalvikVM**[1] に実装されている GC の動作を調査した．その結果，オブジェクト間に存在する参照を辿って各オブジェクトを探索し，到達したオブジェクトへマークを施す処理に時間を多く要していることが分かった．また，既にマーク済みのオブジェクトに対しても繰り返しマーク処理が適用されていることも分かった．そこで本稿では，マーク済みのオブジェクトを管理するための専用表をプロセッサに追加し，オブジェクトの探索時にこの表を参照することで，マーク済みのオブジェクトに対する冗長なマーク処理を省略する手法を提案する．これにより，GC 実行時間の多くを占めるオブジェクトの探索処理を高速化し，GC の高速化を実現する．

2. 研究背景

本章では研究背景として，GC とその代表的アルゴリズム，および GC の高速化に関する既存研究について述べる．

2.1 ガベージ・コレクション

GC とは，プログラムが動的に確保したメモリ領域のうち，不要に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メモリ領域の概念図を**図 1** に示す．なお，図中の矢印は各オブジェクト間の参照関係を表している．ヒープ領域内に配置されたオブジェクトを指すポインタは，グローバル変数やコールスタック，レジスタなど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**と呼び，これを起点としてポインタを辿ることで，ヒープ領域内のオブジェクトを参照することができる．また，ヒープ領域内に

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

© 2016 Information Processing Society of Japan

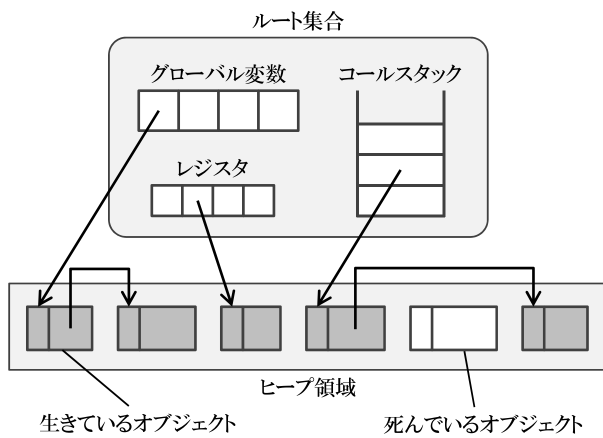

図 1　プログラム実行時のヒープ領域と参照関係の様子

配置されたオブジェクトは他のオブジェクトへのポインタを保持することもある．そのような他のオブジェクトから参照されているオブジェクトは，ルート集合から複数のオブジェクトを経由することで参照できる．このようにルート集合から直接，あるいは間接的に参照可能なオブジェクトを**生きているオブジェクト**と呼ぶ．一方，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**と呼ぶ．GC はこの死んでいるオブジェクトを破棄し，これに割り当てられていたメモリ領域を解放することで，不要なオブジェクトに割り当てられていたメモリ領域を再利用できるようにする．

GC の代表的なアルゴリズムの一つに **Mark & Sweep**[2] がある．このアルゴリズムは，ルート集合からポインタを辿り，生きているすべてのオブジェクトにマークを付けるマークフェーズと，マークの付けられなかった死んでいるオブジェクトを回収するスイープフェーズの二つのフェーズで構成されており，これらのフェーズを交互に繰り返しながら動作する．

また，他にも GC の代表的なアルゴリズムとして，Copying[3] と Reference Counting[4] があり，現在研究されているすべての GC アルゴリズムは，これら 3 つの基本的アルゴリズムの組み合わせ，もしくはその改良であることが知られている [5]．特に Mark & Sweep は実装が比較的容易であることから，多くのアルゴリズムのベースとなっている．

2.2　関連研究

GC の代表的な改良アルゴリズムとして **Concurrent GC**[6] が挙げられる．Concurrent GC は，GC をアプリケーションと並行に動作させることで，システムの停止時間を短縮することを目的としたアルゴリズムである．なお，GC は他のアプリケーションと並行動作する場合，オブジェクトの探索中にアプリケーションによるポインタの書き換えが発生することで，マーク漏れを引き起こす可能性がある．そこで Concurrent GC では，バリア同期を用いることでマーク処理中のポインタ書き換えを検知し，そのポインタから改めてマーク処理を開始することで，マーク漏れの発生を防ぎ，アプリケーションとの並行動作を可能としている．しかし，GC とアプリケーションとの間で必要となる同期処理などのオーバヘッドによって，アプリケーションのスループットが低下してしまうというデメリットがある．

GC の高速化は以上で述べたような，ソフトウェア面における改良がほとんどであるが，わずかながらハードウェア支援による既存研究も存在している．その例として，SILENT[7] や Network Attached Processing（NAP）[8] がある．これらは Concurrent GC 同様，GC とアプリケーションを並行動作させるアルゴリズムを採用しており，その際必要となるバリア同期をハードウェアサポートすることでオーバヘッドを抑制し，高速化を図っている．

3.　GC の動作解析

本稿では代表的な GC のアルゴリズムを構成する処理要素のうち，ボトルネックとなっている処理をハードウェア支援することで，GC の大幅な高速化を目指す．

3.1　GC の処理要素ごとの内訳調査

GC のボトルネックを調査するために，代表的な GC アルゴリズムの動作を解析する．なお，本調査はモバイル端末の実行環境として代表的な DalvikVM に実装されている Mark & Sweep を対象として行う．まず，GC の各フェーズに要する実行時間の内訳を調査した．本調査には，フルシステムシミュレータである gem5 [9] を用い，この上で DalvikVM をシミュレート実行することで，GC の各処理に要する実行サイクル数を計測した．なお，DalvikVM 上で実行するプログラムには，GCBench[10]，AOBench[11]，および SPECjvm2008[12] から 5 個の，計 7 個のベンチマークプログラムを用いた．その結果，マークフェーズは GC 処理全体の実行サイクル数に対し最大約 80%，平均約 46% もの実行サイクル数を占めていることが確認できた．

これを踏まえ本稿では，このオブジェクトの探索処理が GC 実行時間の多くを占めている要因を調査し，これをハードウェア支援することで，多くの GC アルゴリズムの高速化を目指す．

3.2　DalvikVM におけるオブジェクト探索

ヒープ領域上のオブジェクトを探索するためには，マークを施したオブジェクトが保持している参照を再帰的に辿る必要がある．これを実現するために，DalvikVM ではマークを施したオブジェクトを管理するための**マークスタック**と呼ばれるスタックを用いている．

このマークスタックには，マークを施したオブジェクトが順にプッシュされる．そして，このスタックからオブ

Vol.2016-ARC-221 No.18
2016/8/9

表 1 同一オブジェクトに対するマーク処理の発生回数

X	1	10	50	100	200
AOBench	345	151	37	20	11
GCBench	61,717	6,312	1,269	636	319
crypto.rsa	2,789	1,418	344	178	93
crypto.aes	2,804	854	218	115	61
crypto.signverify	2,801	1,412	349	180	94
compress	627	321	84	45	25
serial	4,269	1,156	313	165	87

ジェクトをポップすると同時に，当該オブジェクトが参照しているオブジェクトを探索し，同様にこれが参照しているオブジェクトへのマーク，およびスタックへのプッシュを行う．以上の操作を，マークスタックが空になるまで再帰的に繰り返していくことで，ヒープ領域上の全てのオブジェクトを探索できる．

しかし，マークを施したオブジェクトを全てマークスタックにプッシュするような単純な実装では，オブジェクト間に循環参照が存在する場合に対応できない．そこでDalvikVMでは，オブジェクトがマーク済みか否かを記憶してあるMarkビットマップと呼ばれるビット列を，各オブジェクトへのマーク時に確認し，当該オブジェクトに対応するビットがセットされている場合，つまり当該オブジェクトが既にマークされている場合には，これをマークスタックへプッシュしないようにしている．

しかし，このようにマークスタックへプッシュするか否かをその直前に判断するため，探索済みのオブジェクトが繰り返しマークスタックにプッシュされることはなくなるものの，それらのオブジェクトに対するマーク処理自体は毎回実行されてしまう．本来，各オブジェクトへのマーク処理は一回ずつで十分であり，そのようなマーク済みのオブジェクトに対するマーク処理は本質的には必要のないものである．

そこで，3.1節で述べた調査と同じ環境において，そのような冗長なマーク処理が実際にどの程度発生しているかを調査した．なお本調査では，各ベンチマークプログラムの実行において，最初に発生するGCが完了する時点までに各オブジェクトがマークされた回数を計測した．マーク回数が上位のオブジェクトに対し，その平均回数をまとめた結果を**表 1**に示す．なお表中のXは，平均を求めたオブジェクトの個数を示しており，例えば$X = 10$の欄はマーク回数が多かった上位10個のオブジェクトに対する平均マーク回数を示している．

結果を見ると，オブジェクトの中には千回以上マークされているものも含まれていることが分かる．これは，3.1節で調査した結果において，マーク処理に要する時間が長くなる原因にもなっていると考えられる．また，crypto.rsaとcrypto.signverifyの行を見ると，$X = 1$の場合のマーク回数の平均に対する$X = 10$の場合のマーク回数の平均の比率が，他のベンチマークプログラムより大きいことから，これらのベンチマークプログラムでは，頻繁に冗長なマークがなされるオブジェクトが特に多く存在すると考えられる．そのためこの冗長なマーク処理を省略することで，マーク処理の高速化，およびこれに伴うGCの性能改善を実現できると考えられる．

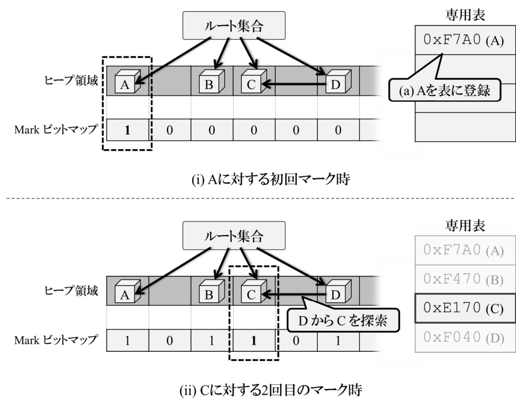

図 2 専用表を用いたマーク処理の省略

4. 専用表を用いた冗長なマーク処理の省略

本章では，前章で述べた冗長なマーク処理を省略するGC高速化手法を提案し，その概要と動作モデルについて順に述べる．

4.1 提案手法の概要

前章で述べたGC処理のオーバヘッドを削減するために，本稿では冗長なマーク処理を省略するためのハードウェア支援手法を提案する．本節では，本提案手法の概要，および提案手法におけるGCの動作イメージについて述べる．

GC実行時の冗長なマーク処理を省略するために，マーク対象のオブジェクトが過去にマークされたことがあるか否かをマーク処理前に判断可能とする機構を新たに実装する必要がある．これを実現するために，本提案手法ではプロセッサのハードウェアを拡張し，マーク済みのオブジェクトを記憶するための専用表を追加する．さらに，これに併せて既存のマークフェーズの動作を拡張し，各オブジェクトへのマーク処理を，この専用表の確認後に行うように変更する．そして，マーク対象のオブジェクトが既に表に登録されている場合，つまり当該オブジェクトが既にマークされている場合，これに対するマーク処理を省略することで冗長なマーク処理を防ぐ．一方，マーク対象のオブジェクトが表に登録されていない場合，今後の同一オブジェクトに対するマーク処理を省略するために，これを表に登録しておく．

ここで，この専用表を用いた場合のマーク処理の省略手順を**図 2**に示す．この図は，ヒープ領域上に存在する4個のオブジェクトAからDに対して，順にマーク処理を施

© 2016 Information Processing Society of Japan

す例を示している．まずオブジェクト A に対して初めてマークを施す際 (i)，A はまだ専用表に登録されていないため通常通りマーク処理を行う．これと同時に，提案手法では以降 A に対する冗長なマーク処理を省略するために，A に割り当てられているヒープ領域のアドレスを専用表へ登録しておく (a)．そしてこの動作を，探索する全オブジェクトに対して繰り返していくことで，やがて専用表には A から D のオブジェクトが登録されることになる．そのため，例えば (ii) に示すように，D から C への参照を辿って再度 C を探索する際，C は既に専用表に登録されているため，この C に対する冗長なマーク処理を省略する．

4.2 エントリの管理方法

既存の DalvikVM において大きなオーバヘッドとなっているオブジェクト探索処理を高速化するために，前節で述べたように本提案手法ではマーク済みのオブジェクトを専用表で管理する．これにより，冗長なマーク処理を省略する．なお，この冗長なマーク処理を完全に排除するためには，マーク済みの全てのオブジェクトを専用表で管理する必要がある．しかし，管理すべきオブジェクトの数はプログラムによって異なり，これらを十分に記憶するために専用表を大きくすると，追加ハードウェア量が非常に大きくなってしまうことによる，回路面積や消費電力の増加が懸念される．

これを解決する一つの方法として，専用表に登録されている各オブジェクトを LRU ベースの追い出しアルゴリズムを備えたリスト形式で管理することが考えられる．これにより，頻繁にマークがなされるオブジェクトを優先的に，かつ比較的少量のハードウェアで管理できる．

しかし，マーク対象となったオブジェクトを常にリストの先頭へ挿入するような単純な LRU 方式の場合，本来管理すべき，冗長なマーク処理が多く発生するオブジェクトを，優先的に管理できない可能性がある．例えば，冗長なマーク処理がなされるオブジェクトが登録された後に，一度しかマークされないオブジェクトが連続してリストへ登録されることで，先に登録された本来管理されるべきオブジェクトの情報が追い出されてしまう可能性がある．

そこで提案手法では，過去に重複してマークがなされているかどうかに応じて，各オブジェクトを LRU 方式の追い出しアルゴリズムを採用した二つの専用表で管理する．なお本稿では，これらの専用表をそれぞれ**一次検索表**，**二次検索表**と呼ぶ．そしてオブジェクトへのマーク処理時には，一次検索表，二次検索表の順にこれらの専用表を検索し，当該オブジェクトへのマーク処理が省略可能かどうかを判断する．なお，これらの専用表のうち，二次検索表は新たにマーク対象となったオブジェクトを管理するために利用する．そして，二次検索表の中で再度マーク対象となったオブジェクトのみ，一次検索表を用いて管理する．

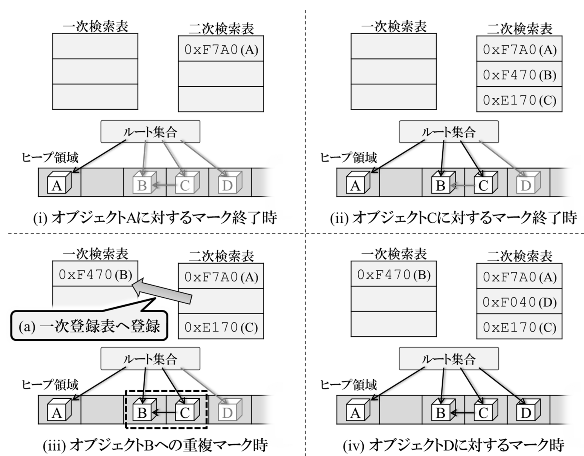

図 3 二つの表を用いたエントリの管理

これにより，重複してマークがなされるオブジェクト，つまり冗長なマーク処理が GC 処理の大きなオーバヘッドになっているオブジェクトを，一次検索表で優先的に管理できるようになる．

ここで，以上で述べた二つの専用表を用いたエントリの管理方法を**図 3** に示す例を用いて説明する．この例では，オブジェクト A から D を順に探索する例を示しており，各表のエントリ数は 3 と仮定している．まずオブジェクト A に対してマークを施す際 (i)，これは A に対する初回のマーク処理であるため，オブジェクト A のアドレスを二次検索表に登録する．この動作をオブジェクト B と C に対しても繰り返すことで，やがて二次検索表には (ii) に示すように A から C の三つのオブジェクトが登録される．ここで，二次検索表に登録済みのオブジェクト B が再度探索された場合 (iii)，これを二次検索表から削除して一次検索表へと登録する (a)．このような動作により，重複してマークがなされたオブジェクトのみを，優先的に一次検索表で管理する．なお，この状態でオブジェクト D をマークした場合 (iv)，これを二次検索表に登録することで二次検索表の全てのエントリにオブジェクトが登録されることになる．このような状態で，他のオブジェクトが新たにマーク対象となった場合には，二次検索表の中から古いエントリを追い出すようにする．

5. 冗長なマーク処理省略手法の実装

本章では，前章で述べた冗長なマーク処理を省略する手法の具体的な実装について述べる．

5.1 専用表の構成

本節ではまず，マーク済みのオブジェクトを記憶するための各専用表の具体的な構成について述べる．

5.1.1 一次検索表

本提案手法では，4.2 節で述べたような LRU に基づく追い出しアルゴリズムを備えた二つの専用表を用いるこ

© 2016 Information Processing Society of Japan

とで各オブジェクトを管理する．これにより，表のエントリがあふれた際に，マークされてからの経過時間が最も長いオブジェクトを適宜専用表から追い出すようにする．2つの検索表のうち，高頻度でマーク対象となっているオブジェクトが登録されているのは一次検索表であるため，この一次検索表に登録されているオブジェクトに対する冗長なマーク処理を省略する．これを実現するためには，マーク処理の際に対象オブジェクトが一次検索表に登録されているか否かを，できるだけ小さなレイテンシで確認する必要がある．そこで，一次検索表は高速な連想検索が可能な汎用 CAM（Content Addressable Memory）を用いて実装する．

一次検索表は，マーク済みのオブジェクトに割り当てられているヒープ領域のアドレスを保持する Address，マークされてからの経過時間に基づいてオブジェクトを順序付けした LRU リストにおいて，各オブジェクトの前後に存在するオブジェクトのインデクスを記憶する prev，next の三つのフィールドで構成される．なお，prev，next の各フィールドは一次検索表内でそれぞれ該当するオブジェクトが格納されている表のインデクス番号を保持する．さらに，リスト先頭へのエントリの挿入や，末尾のエントリの追い出しを実現するために，一次検索表が管理するリストの先頭，および末尾のオブジェクトに付与されたインデクス番号を保持するレジスタ（Head，Tail）と，現在保持しているオブジェクト数を管理するためのレジスタ（#Addr）を追加する．

5.1.2 二次検索表

二次検索表は，マーク対象のオブジェクトが一次検索表に登録されていなかった場合に参照する．そして，当該オブジェクトが二次検索表に登録されていた場合は，これを一次検索表へ移し替える．二次検索表に対するこの操作は，マーク対象オブジェクトを一次検索表から検索したのち，一次検索表に登録されていなかった当該オブジェクトに対する処理である．よってそれ自体にかかるレイテンシが GC 性能に与える影響は小さい上，当該オブジェクトに対するマーク処理と並行して行うことで隠蔽することができる．よって，高速な連想検索が可能ではあるが回路面積や消費電力の面で劣る CAM ではなく，RAM を用いて二次検索表を実装する．しかし RAM を用いた場合，単純な検索処理では表の全エントリに対してシーケンシャルにアクセスする必要があり，検索コストが非常に大きくなってしまう．そこで本提案手法では，二次検索表のデータ構造としてセットアソシアティブ方式を採用し，ハッシュを利用することで検索コストを抑制する．

二次検索表へオブジェクトを登録する際には，これを登録するセットを決定するために，まず当該オブジェクトのアドレス値を用いてハッシュ値を計算する．そして，各セットに対しウェイ数以上のオブジェクトが登録された際には，いずれかのエントリを新しいエントリで上書きできるようにする．このために，次に上書きされるエントリを記憶しておくためのフィールド（Victim Index）をセット毎に用意する．なおこのフィールドは，ウェイ番号の最大値と等しい値までをカウントできるリングカウンタで構成する．そして，オブジェクトを登録する際は，この値が指すウェイのエントリを上書きする．

一方，オブジェクトのアドレスをキーとして二次検索表を検索する際には，オブジェクトの登録時と同様，まず当該オブジェクトのアドレス値を用いてハッシュを計算し，これに対応するセットを特定する．そして，特定したセットの全エントリに格納されているアドレス値と一致比較することで，検索対象のオブジェクトが二次検索表に登録されているかどうかを確認する．なお，この時に発生する一致比較は，最大でもウェイ数と等しい回数のみであり，表の全エントリに対してシーケンシャルにアクセスする場合と比較して，一致比較に要するコストを大幅に抑制できる．

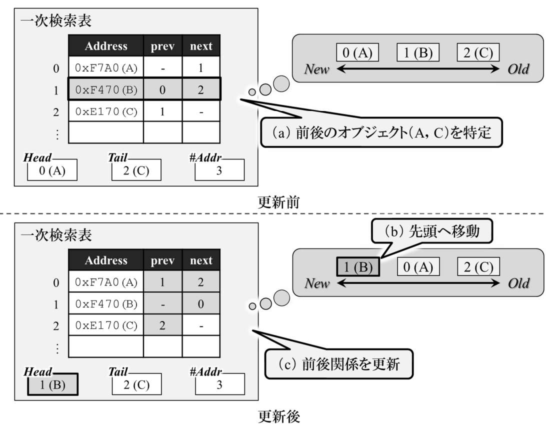

図 4　一次検索表に登録されている場合の操作

5.2 専用表に対する操作

本節では，これまでに述べた専用表に対する操作を，マーク対象のオブジェクトが一次検索表に登録されている場合と，そうでない場合の二通りに分けてそれぞれ示す．

5.2.1 一次検索表に登録されている場合

提案手法では，マーク対象のオブジェクトが一次検索表に登録されている場合，つまり当該オブジェクトがすでに複数回マーク対象となっている場合，これに対するマーク処理を省略する．そして LRU に基づき，当該オブジェクトに対応するエントリをリストの先頭に挿入する．

ここで，図 4 を用いて，この時の一次検索表に対する具体的な操作について述べる．この図は，一次検索表に登録済みのオブジェクト B が再度マークされた場合における操作を示している．この時，B に対するマーク処理を省略した後，これをリストの先頭に移動するために，まず B の前後に配置されているオブジェクト A と C を特定する (a)．その後，B をリストの先頭に移動し (b)，これに伴って一

© 2016 Information Processing Society of Japan

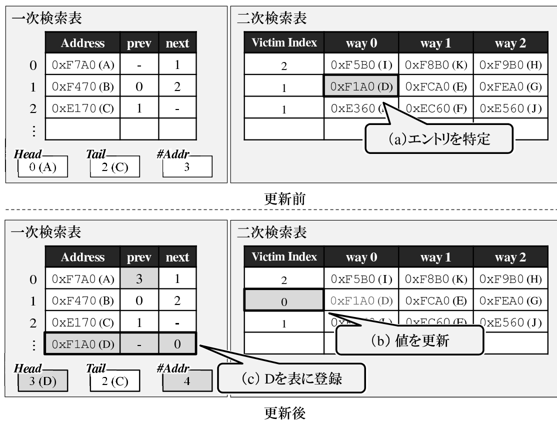

図 5　二次検索表に登録されている場合の操作

次検索表の内容，およびレジスタの値を更新する (c)．この例では，B を先頭に移動したことに伴い，A と C が隣接するように一次検索表 の内容を更新する必要がある．そこでまず，A に対応するエントリの next を，C に付与されたインデクス番号である 2 へと更新する．同様に，C に対応するエントリの prev を，A に付与されたインデクス番号である 0 へと更新する．その上で，リストの先頭のオブジェクトを示すレジスタ Head の値，および B の前後関係を更新する．

5.2.2　一次検索表に登録されていない場合

マーク対象のオブジェクトが一次検索表に登録されていない場合には，これに対して通常のマーク処理を施すと同時に，当該オブジェクトのアドレスからハッシュ値を計算し，これを用いて二次検索表を検索する．そして二次検索表に登録済みである場合には，当該オブジェクトに対する以降の冗長なマーク処理を省略するために，これを一次検索表に登録する．ここで，この時の各専用表に対する操作を**図 5** に示す．この図は，二次検索表に登録済みのオブジェクト D が再度マークされた際の各専用表に対する操作を示している．まず，D のアドレスからハッシュ値を計算し，これが格納されているセットを選択する．そして，そのセット内で一致比較することで，D が格納されているエントリを特定する (a)．次に，この D を一次検索表へ移し替えるために，D を二次検索表から削除する．なお，D を削除したことに伴い，これが格納されていたエントリが空くため，このエントリを持つセットの Victim Index の値を D が格納されていたエントリのウェイ番号に更新する (b)．これにより，当該セットに対して次にオブジェクトが登録される際に，この空きエントリが登録先となるようにする．その後 D を一次検索表に登録し (c)，D の前後関係，および各レジスタの値を更新する．

ここで，以上で述べた動作によって一次検索表にオブジェクトを登録する際，一次検索表のエントリが溢れてしまう場合がある．その場合，一次検索表が管理するリストの末尾のオブジェクトを追い出すことでエントリを確保する．なお，一次検索表から追い出したオブジェクトは二次検索表に登録し，当該オブジェクトが再度マーク対象となった場合に対応できるようにする．

一方，マーク対象のオブジェクトが二次検索表にも登録されていない場合は，これを二次検索表に登録するために，まず当該オブジェクトのアドレスから求めたハッシュ値を用いて対応するセットを特定し，Victim Index の値を取得する．そして，この値に対応するウェイに当該オブジェクトを登録し，Victim Index の値をインクリメントする．

表 2　シミュレータ構成

Platform	ARM-RealView PBX
Processor	ARMv7
Clock	2.0 GHz
Memory	256 MB
OS	Linux 2.6.38.8-gem5

6.　評価

本章では，提案手法をシミュレーションにより評価し，得られた結果から提案手法の有用性について考察する．また，提案手法を実装する上で必要となるハードウェアコストの見積りも示す．

6.1　評価環境

評価にはフルシステムシミュレータである gem5 シミュレータを用いた．本評価で想定するシステムの構成を**表 2** に示す．DalvikVM 上で実行するベンチマークプログラムには，GCBench[10]，AOBench[11]，また汎用ベンチマークプログラムである SPECjvm2008[12] から 5 個の，計 7 個を使用した．なお，本評価で使用する gem5 シミュレータはフルシステムシミュレータであり，ベンチマークプログラム以外のプログラムも同時に実行されている．また，それらのプログラムの実行状態は毎回異なり，ベンチマークプログラム実行時に利用可能なリソースの状態も毎回異なるため，性能のばらつきを考慮する必要がある．そこで本評価では各評価対象につき試行を 10 回繰り返し，得られた結果から平均値を算出した．

ここで，本評価で用いた各専用表のエントリ数について述べる．本提案手法では，一次検索表に登録されているオブジェクトに対するマーク処理を省略するため，一次検索表のサイズが大きいほど効果が高くなると考えられる．しかし，3.2 節の表 1 に示した調査結果を見ると，冗長なマーク処理の発生は，マーク回数が特に多い上位数十個のオブジェクトに集中していることが分かる．また，一次検索表を構成する CAM は消費電力が比較的大きいため，そのサイズは可能な限り小容量に抑えることが望ましい．そこで，本評価で使用する一次検索表のサイズは，同様に CAM で構成される TLB が一般的に十数エントリから数十エントリであることも踏まえ，50 エントリとして評価を行った．

© 2016 Information Processing Society of Japan

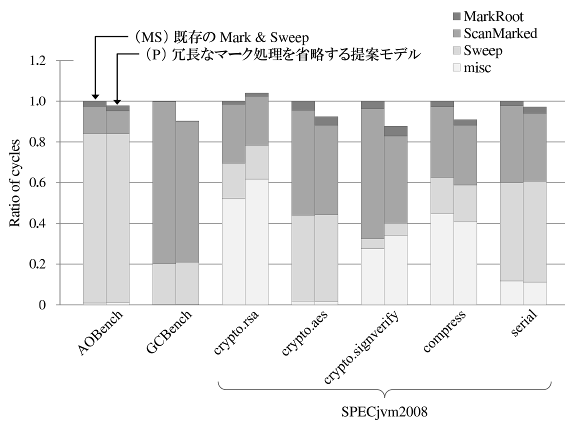

図 6　GC 実行サイクル数

また二次検索表は，頻繁にマークされるオブジェクトを極力追い出さずに管理できることが望ましい．表 1 に示した結果において，$X = 100$ の場合と $X = 200$ の場合のマーク回数の平均を比較すると，すべてのベンチマークプログラムで $X = 200$ の場合は $X = 100$ の場合の約半分になっていることから，上位 100 個から 200 個のオブジェクトに対するマーク回数は少ないと考えられる．そこで，マーク回数の多い上位 100 番目から 200 番目のオブジェクトに対するマーク回数を調査した．その結果，上位 100 番目のオブジェクトはいずれのベンチマークプログラムにおいても複数回マークされていたが，上位 200 番目のオブジェクトは一部のベンチマークプログラムにおいて，一度しかマークされていなかった．また，冗長なマークがなされるオブジェクトが多く含まれるベンチマークプログラムにおいても，上位 200 番目のオブジェクトは最大で 6 回しかマークがなされていないことがわかった．このことから，二次検索表を 200 エントリ程度で実装すれば，頻繁にマークされるオブジェクトを管理するのに十分であると考えられる．そこで，本評価で使用する二次検索表のサイズは，3 ウェイ 64 セット構成の 192 エントリとして評価を行った．

6.2　評価結果

本評価では，まず提案手法が GC に及ぼす影響を評価するために，既存手法と提案手法の GC 実行サイクル数を計測した．さらに，GC がシステムの全体性能に及ぼす影響を評価するために，GC によるシステムの平均停止時間も計測した．

6.2.1　GC の実行サイクル数

まず，GC 全体の実行サイクル数の計測結果を図 **6** に示す．図 6 では，各ベンチマークプログラムの結果を 2 本のグラフで示している．これらのグラフはそれぞれ，

(MS)　既存の Mark & Sweep を実行するモデル
(P)　冗長なマーク処理を省略する提案モデル

の各モデルにおいて GC の実行に要したサイクル数を示しており，既存モデル (MS) の GC 実行サイクル数を 1 として正規化している．また，凡例はグラフの内訳を示しており，MarkRoot はルート集合から直接参照されているオブジェクトへのマークに要したサイクル数，ScanMarked はマーク済みのオブジェクトが持つポインタを辿り，それらが参照しているオブジェクトを探索するために要したサイクル数，Sweep はスイープ処理に要したサイクル数，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．

評価結果を見ると，既存モデル (MS) と比較して提案モデル (P) は crypto.rsa を除く全てのベンチマークプログラムで GC の実行サイクル数を削減できていることが分かる．これは，提案手法によって冗長なマーク処理が省略され，ScanMarked が削減されたためである．特に，crypto.signverify では ScanMarked の占める割合が大きく，提案手法が有効に働いたため，12.4%の GC 実行サイクル数が削減できている．またベンチマークプログラム全体でも，平均 5.7%の GC 実行サイクル数を削減できていることを確認した．

なお，crypto.rsa，crypto.signverify，compress の結果を見ると，Mark & Sweep 以外の処理に要したサイクル数である misc が増加，もしくは減少していることが分かる．これは，これらのベンチマークプログラムでは，GC の実行回数が他のベンチマークプログラムと比較して少なく，GC 実行サイクル数そのものが非常に少ないため，ベンチマークプログラム以外の，並行動作しているプログラムによるリソース使用の影響が大きく現れてしまったためだと考えられる．しかし，いずれのベンチマークプログラムでも ScanMarked は削減されており，提案手法によってオブジェクトの探索処理を高速化できたことに変わりはない．

なお，提案手法では GC 実行サイクル数を削減できる一方，専用表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，一次検索表のアクセスレイテンシを 2cycles，二次検索表のアクセスレイテンシを 1cycle と仮定してオーバヘッドを概算したところ，GC 実行サイクル数に対するオーバヘッドの比率は，提案モデル (P) で平均約 1.8%となり，十分に小さいものであることが確認できた．また 5.1.2 項でも述べたように，専用表に対する操作の一部は通常のマーク処理と並行して行うことが可能である．そのため，専用表の操作に要するコストはある程度隠蔽可能であり，実質的なオーバヘッドはさらに小さくなると考えられる．

6.2.2　GC による平均停止時間

次に GC による平均停止時間の評価結果を図 **7** に示す．なお本評価では，GC の実行によってアプリケーションが停止していた時間の総和を GC 実行回数で除算することで，GC 実行毎の停止時間の平均を算出した．図 7 では各ベンチマークプログラムの結果を図 6 で示した二つのモデルに

© 2016 Information Processing Society of Japan

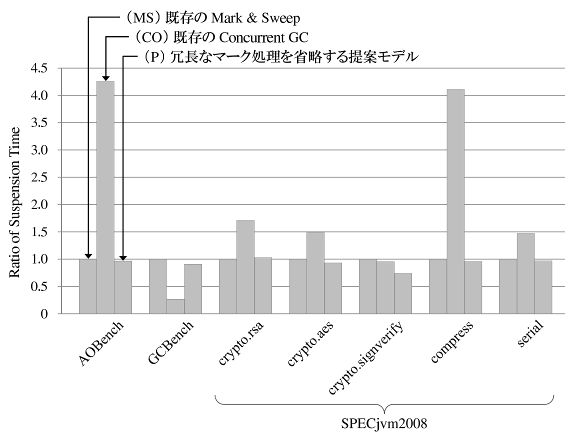

図 7　GC による平均停止時間

(CO)　既存の Concurrent GC を実行するプログラム

を加えた三つのモデルの結果を示しており，既存モデル (MS) の停止時間を 1 として正規化している．

評価結果を見ると，提案モデル (P) では多くのベンチマークプログラムで停止時間を削減できていることがわかる．これは，提案手法により GC 一回に要する実行サイクル数が削減されたためである．一方 Concurrent GC (CO) は，2.2 節で述べたようにスループットをある程度犠牲にしつつも，停止時間を短縮することを目的とした手法であるが，AOBench や compress では却って停止時間が大きく増加してしまっている．これは，これらのプログラムでは GC 一回あたりに要する時間が短く，同期処理などのコストが相対的に大きくなってしまったためだと考えられる．しかしこのようなプログラムについても，提案モデルは停止時間の増大を防ぐことができている．ベンチマークプログラム全体では，提案モデルを用いた場合，平均停止時間を最大で約 26.0%，平均で約 7.1%短縮できることが確認できた．

6.3　ハードウェア構成の妥当性

本提案手法では 5.1.2 項で述べた通り，二次検索表をセットアソシアティブ構成とすることで，この表へのアクセスに要するオーバヘッドを抑制している．また，二次検索表の各セットをラウンドロビンで管理することで，ある程度古いオブジェクトから順に追い出すことを可能としている．しかしこのようなエントリの管理方法では，一次検索表のように完全な LRU を用いた場合と比較して，今後頻繁にマークされると考えられるオブジェクトを多く追い出してしまっている可能性がある．そこで，二次検索表をキャッシュアソシアティブ構成ではなく，一次検索表と同様にリスト構造を用いて完全な LRU で管理したと仮定したモデルを，提案モデルと比較評価した．その結果，これらの GC 実行サイクル数削減率にはほとんど差異がないという結果が得られた．

この理由を考察するにあたり，一次検索表に対する検索ヒット率を調査した．その結果ベンチマークプログラム全体では一次検索表に対する検索ヒット率は 50%程度となった．一方，全てのオブジェクトを登録可能な十分なサイズを一次検索表に想定した場合でも，ヒット率は約 62%となることを確認した．このことから，本評価で想定した 50 というエントリ数でも，冗長なマーク処理を十分抑制できていると考えられる．

6.4　ハードウェアコストの見積り

本節では，提案モデルで必要となるハードウェアコストについて考察する．6.1 節で述べた通り，提案モデルでは 50 エントリの一次検索表と，3 ウェイ 64 セット構成の二次検索表を用いる．このうち一次検索表における各フィールドのビット幅は，マーク済みのオブジェクトのアドレスを記憶するために 32bit，前後のオブジェクトを記憶するためにそれぞれ 8bit ずつ必要である．また，リストの先頭，および末尾に配置された各オブジェクトと，一次検索表 が保持しているオブジェクト数を管理するためのレジスタに必要なビット数は，それぞれ 8bit である．以上より，一次検索表は 300Byte の CAM，および三つの 8bit レジスタで構成できる．一方で二次検索表は，登録先のウェイを決定するカウンタと，マーク済みのオブジェクトのアドレスをウェイ数分保持するためのフィールドから構成される．本提案手法では，ウェイ数を 3 としているため，カウンタの幅は 2bit である．そのため，二次検索表の各セットに必要なビット数は，カウンタ値を記憶するために 2bit，マーク済みのオブジェクトのアドレスを 3 ウェイ分記憶するために $32 \times 3 = 96$bit 必要である．以上より，二次検索表は 784Byte の RAM で構成できる．

これらのことから，本提案手法のハードウェアコストは合計でも約 1KByte となり，少量のハードウェアで実現できることが確認できた．

7.　おわりに

本稿では，多くの GC アルゴリズムで必要となるオブジェクト探索処理をハードウェア支援によって高速化する手法を提案した．この手法では，マーク済みのオブジェクトを記憶するための専用表をプロセッサに追加し，GC 実行時にこれを参照することで，従来の冗長なマーク処理に要していたコストを削減し，GC の高速化を実現した．

提案手法の有効性を確認するため，シミュレーションによる評価を行った．その結果，既存の Mark & Sweep と比較して，GC サイクル数を最大 12.4%削減できることが分かった．また，Concurrent GC では一部のベンチマークプログラムにおいてスループットが悪化したり，停止時間が長くなってしまう場合があるのに対し，提案手法ではそのような性能悪化を抑制できることを確認した．

本研究の今後の課題として，既存のアルゴリズムに囚わ

© 2016 Information Processing Society of Japan

れない，ハードウェア支援を前提とした新たな GC アルゴリズムを考察することが挙げられる．本稿で述べた手法では，GC 処理のうちオブジェクト探索処理のみを高速化している．そのため，プログラムによってはこの手法の効果があまり得られない可能性がある．そこで，追加ハードウェアを利用するために最適化した GC アルゴリズムを考察することで，GC の飛躍的な性能向上の方法を模索していきたい．

参考文献

[1] Bornstein, D.: Dalvik Virtual Machine Internals, Google I/O 2008 (2008).
[2] McCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, *Communications of the ACM*, Vol. 3, pp. 184–195 (1960).
[3] Minsky, M.: A LISP Garbage Collector Algorithm Using Serial Secondary Storage, Technical report, Massachusetts Institute of Technology (1963).
[4] Collins, G. E.: A Method for Overlapping and Erasure of Lists, *Communications of the ACM*, Vol. 3, pp. 655–657 (1960).
[5] 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).
[6] 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).
[7] Takeuchi, I., Yamazaki, K., Amagai, Y. and Yoshida, M.: Lisp can be "Hard" Real Time, *Proc. Japan Lisp User Group Meeting (JLUGM)* (2000).
[8] 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).
[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] Boehm, H.: An Artificial Garbage Collection Benchmark, http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench.html.
[11] Fujita, S.: Ambient Occlusion Benchmark, http://code.google.com/p/aobench/.
[12] SPEC.: SPECjvm2008 benchmarks, http://www.spec.org/jvm2008/ (2008).

© 2016 Information Processing Society of Japan