修士論文

ハードウェア支援を用いた
冗長なマーク処理の抑制によるGC高速化手法

Hardware Support
for Avoiding Redundant Marking
in common Garbage Collections

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学大学院 工学研究科
修士課程 創成シミュレーション工学専攻
平成 25 年度入学　25413508 番

井手上 慶

平成 27 年 2 月 4 日

ハードウェア支援を用いた
冗長なマーク処理の抑制による GC 高速化手法

井手上 慶

内容梗概

スマートフォンに代表されるモバイル機器の普及に伴い，これらの機器への性能要求が高まりを見せている．このようなモバイル機器では一般に，搭載されるメモリの容量が汎用計算機と比較して少ないため，ガベージ・コレクション（GC）のようなメモリ管理機能の性能が非常に重要となる．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となった領域を自動的に解放する機能であり，これは現在，その有用性から多くの言語処理系に実装されている．

しかし GC には，GC 実行時のアプリケーションプロセス停止に伴い，システムのレスポンスが低下してしまうという問題が存在する．この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る．そのため，こうした既存の改良アルゴリズムは GC が抱える問題の根本的な解決策とは成り得ていない．一方で，GC をハードウェア支援によって高速化しようとする試みも行われてきたが，それらは特定の言語における GC 実装で必要となる同期処理のみを高速化するものであるなど，その適用範囲は非常に狭い．

これに対し本研究では，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で，GC の大幅な高速化と，高速化手法自体の高い汎用性の実現を図る．また，GC を高速化することでクロック周波数を大きく向上させずとも高い性能を実現できるようになり，これに伴って消費エネルギーも抑制できると考えられる．特にバッテリ駆動が前提となるモバイル機器においては，消費エネルギーの抑制は駆動時間の延長に繋がるものであり，これに伴って可用性が大きく向上することも期待できる．

これらを実現するにあたり，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 を対象として，GC を構成する処理の実行時間の内訳を調査し，動作のボトルネックを考察した．その結果，DalvikVM では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要していることが分かった．この要因を調査したところ，DalvikVM におけるオブジェクトの探索処理では過去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を施しており，この冗長なマーク処理が GC の大きなオーバヘッドになっていることが分かった．

そこで本論文では，マーク済みのオブジェクトを専用の表に記憶しておき，オブジェクトの探索時にこれを参照することで，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 の大幅な高速化を実現する．なお，マーク処理の省略によって得られる効果を高めるべく，本提案手法ではマーク済みのオブジェクトを記憶するために二つの専用表を組み合わせて利用する．過去にマークがなされたオブジェクトを，マーク処理が複数回実行されているかどうかに応じてこれら別々の専用表を用いて管理することで，冗長なマーク処理が頻発しているオブジェクトを優先的に管理し，それらのオブジェクトに対するマーク処理を省略することで，マーク処理の大幅な高速化を実現する．

提案手法の有効性を検証するため，これを gem5 シミュレータに実装し，GCBench, AOBench，および SPECjvm2008 を用いて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，最大 12.4%の GC 実行サイクル数を削減できることを確認した．また，GC の代表的な改良アルゴリズムである 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や停止時間が悪化していた一方，提案手法ではそのような性能悪化を抑制できており，手法の有効性を確認した．

ハードウェア支援を用いた
冗長なマーク処理の抑制による GC 高速化手法

目次

1 はじめに

スマートフォンに代表されるモバイル機器の普及に伴い，これらの機器への性能要求が高まりを見せている．このようなモバイル機器では一般に，搭載されるメモリの容量が汎用計算機と比較して少ないため，**ガベージ・コレクション（GC: Garbage Collection）**のようなメモリ管理機能の性能が非常に重要となる．GC とは，プログラムが動的に確保したメモリ領域のうち，不要となった領域を自動的に解放する機能である．GC を利用することで，メモリリークなどのメモリ管理に関連するバグの発生を防ぐことができ，プログラマの負担を軽減できる．そのため，アプリケーション開発の生産性と品質を向上させる目的で，GC を搭載したプログラミング言語が広く開発・利用されてきた．その一方，特にサーバサイド Java 環境などでは，GC がシステム全体に大きな影響を与えるようになったことで，GC の性能が古くから重要視されてきた．これに加えて現在，Android の **DalvikVM**[1] に代表される仮想マシンや，Mozilla の **Firefox OS**[2]，LG の **LuneOS**[3] 等の WebAPI を基幹に用いた実行環境が，モバイル機器をはじめとする様々な機器で多く採用・検討されているが，これらの環境は実行時間に占める GC の割合が大きく，GC が実行環境の性能に大きな影響を与えることが知られている．

このような背景から，GC の性能が与える影響範囲が拡大してきており，今後は GC がシステム全体に大きな影響を与える場面が増加していくと考えられる．しかし GC には，その有用性の一方で，GC 実行時のアプリケーションプロセス停止により，システムのレスポンスが低下してしまうという問題が存在する．この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る．そのため，こうした既存の改良アルゴリズムは GC が抱える問題の根本的な解決策とは成り得ていない．一方で，GC をハードウェア支援によって高速化しようとする試みも行われてきたが，それらは特定の言語における GC 実装で必要となる同期処理のみを高速化するものであり，その適用範囲は非常に狭い．

これに対し本研究では，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で，GC の大幅な高速化と

高速化手法自体の高い汎用性の実現を図る．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へと発展させることで，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．

これを実現するにあたり，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を対象として，GCを構成する処理の実行時間の内訳を調査し，動作のボトルネックを考察した．その結果，DalvikVMでは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要していることが分かった．このオブジェクトの探索は，GCによって回収してはならないオブジェクトを判断するために必要な処理であり，Mark & SweepやCopyingなど多くのGCアルゴリズムで用いられている．しかしDalvikVMでは，オブジェクトを探索する際，過去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を施していることが分かった．そのような冗長なマーク処理は本質的には不要であり，DalvikVMのGCにおける大きなオーバヘッドとなってしまう．

そこで本論文では，マーク済みのオブジェクトを専用の表に記憶しておき，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の大幅な高速化を実現する．なお，マーク処理の省略によって得られる効果を高めるべく，本提案手法ではマーク済みのオブジェクトを記憶するために二つの専用表を組み合わせて利用する．過去にマークがなされたオブジェクトを，マーク処理が複数回実行されているかどうかに応じてこれら別々の専用表を用いて管理することで，冗長なマーク処理が頻発しているオブジェクトを優先的に管理し，それらのオブジェクトに対するマーク処理を省略することで，マーク処理の大幅な高速化を実現する．

以下，2章では研究背景としてGCとその高速化に関する既存研究について述べる．3章では，提案手法のベースとなるDalvikVMに実装されているGCについて述べた後，その動作のボトルネックについて考察する．4章では，ボトルネックとなっている動作をハードウェア支援により高速化する手法を提案し，5章でその具体的な実装について述べる．そして6章で提案モデルの性能を評価し，7章において結論を述べる．

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．

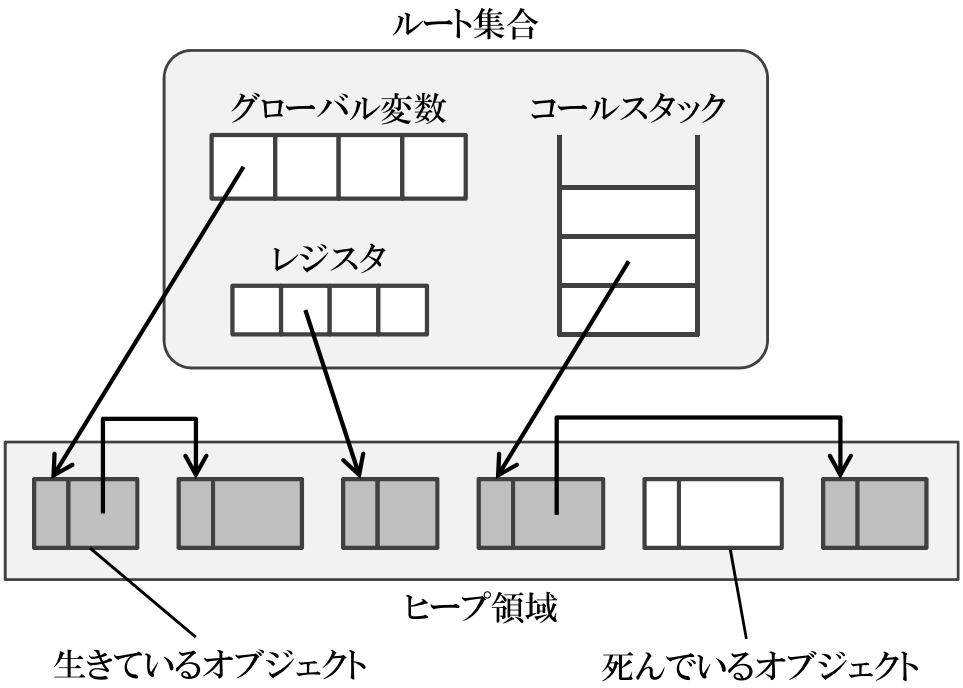

図 1: プログラム実行時のヒープ領域と参照関係の様子

2.1 ガベージコレクション

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．GC を利用することで，プログラマはメモリ管理に労力を割く必要がなくなる．現在 GC は，Java，JavaScript，Lisp，Ruby 等，多くの言語処理系に実装されており，それらのプログラミング言語と実行環境にとって重要な機能の一つとなっている．本節では，まず GC を用いた場合のメモリ管理方法について述べた後，代表的な GC の動作アルゴリズムについて述べる．

2.1.1 GC を用いたメモリ管理

まず，プログラム実行時のメモリ領域の様子の例を**図 1** に示す．プログラム実行時に GC の管理対象となるのは，メモリ領域の中でもヒープ領域内のオブジェクトのみである．なお，ヒープ領域とはプログラムが動的に確保できるメモリ領域のことであり，オブジェクトとはアプリケーションが使用するデータのかたまりのことである．そして生成されたオブジェクトには，ヒープ領域から確保したメモリ領域が割り当てられる．

また，ヒープ領域内に配置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．一方で，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．そして，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．なお，この GC の動作アルゴリズムに関しては様々な研究がなされているが，それらは全て，**Mark & Sweep**[4]，**Copying**[5]，**Reference Counting**[6] という，三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 [7]．以下では，これら三つのアルゴリズムの動作とその特徴について述べる．

2.1.2 Mark & Sweep

Mark & Sweep は，生きている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る．**図 2** はこのアルゴリズムの動作イメージを示しており，図中の (i)，(ii) はそれぞれマークフェーズとスイープフェーズが終了した際のヒープ領域の様子を表している．まずマークフェーズでは，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 (a)．ここで，図中の色付きのオブジェクトはマーク済みのオブジェクトを表している．なお，各オブジェクトは自身のサイズや種類などの情報を保持するヘッダと呼ばれるフィールドを持っており，Mark & Sweep におけるオブジェクトへのマークは，このヘッダの中に確保されたマーク用のビットをセットすることで実現する．そして，このマークフェーズが終了した時点でマークの付いていないオブジェクトは，ルート集合から辿ることのできないオブジェクト，つまりアプリケーションから参照されることのないゴミ（garbage）であると判断できる．そこで，生きている全てのオブジェクトへのマークが完了すると，マークフェーズを終了してスイープフェーズへと移行し，ヒープ領域全体を走査してマークの付いていないオブジェクトに割り当てられたメモリ領域を解放する (b)．

Mark & Sweep のメリットとして，アルゴリズムがシンプルであるため，GC アルゴリズムの中では比較的実装が容易であり，他のアルゴリズムと組み合わせやすいことが挙げられる．一方デメリットとして，プログラム中でオブジェクトに割り当てられるメモリ領域のサイズが統一されていない場合，メモリの確保・解放が繰り返された

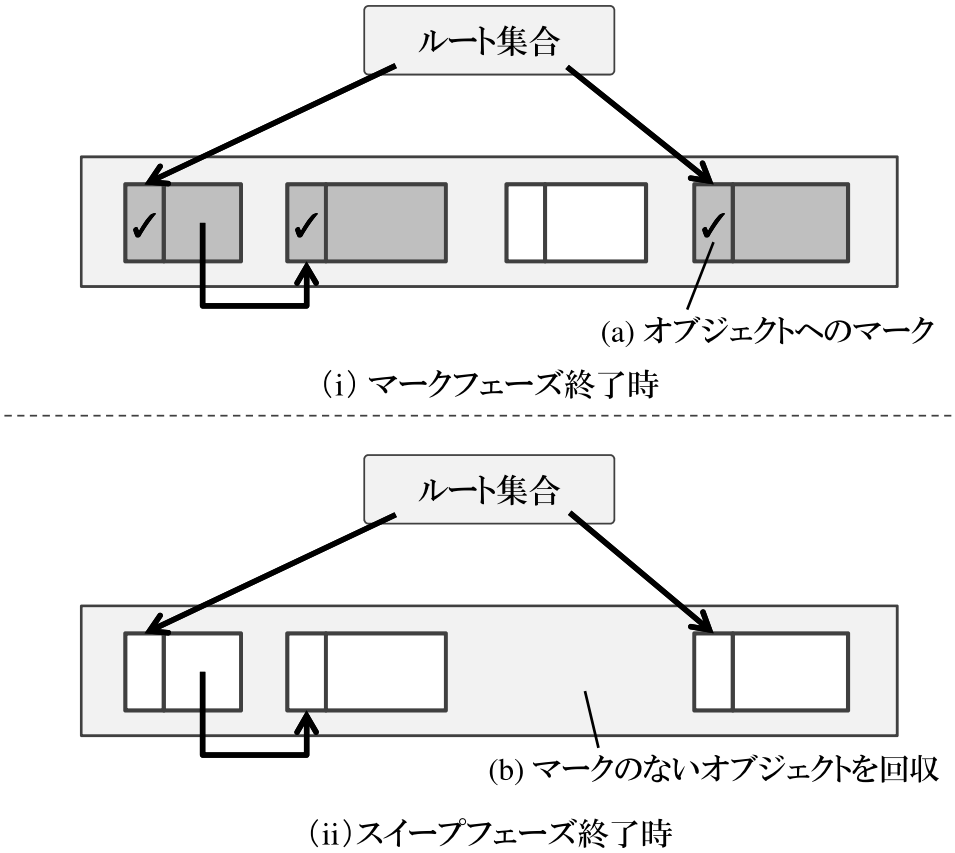

図 2: Mark & Sweep の動作イメージ

結果，ヒープ領域内でフラグメンテーション，すなわち空き領域の断片化が生じる可能性があることが挙げられる．これは，スイープフェーズ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解放するだけで，オブジェクトの移動等は行わないことが原因である．

2.1.3 Copying

Copying は，ある領域上の生きているオブジェクトだけを別の領域にコピーし，元の領域にあったオブジェクトは全て破棄するというアルゴリズムである．このアルゴリズムの動作イメージを**図 3** に示す．Copying では，ヒープ領域をオブジェクトの移動元である from-space と，オブジェクトの移動先である to-space の二つの領域に等分割して使用する．そしてこれらの領域のうち，新たに生成されたオブジェクトには from-space 内に確保したメモリ領域を割り当てる．この際，from-space に十分な空き領域が無く，メモリ領域の割り当てに失敗した場合に GC が実行される (a).

まず，ルート集合からポインタで辿ることのできるオブジェクトを to-space へコピーする．オブジェクトのコピーが全て完了した時点では，ルート集合や to-space 上の各

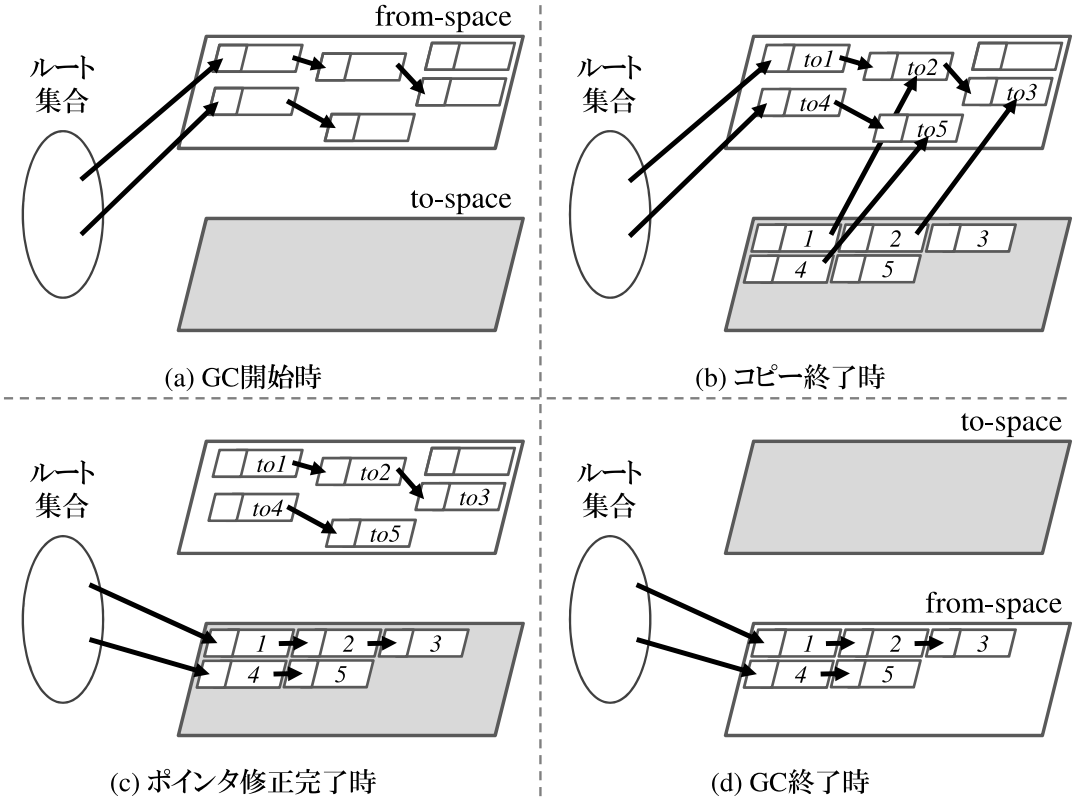

図 3: Copying の動作イメージ

オブジェクトが持つポインタはまだ from-space 上のコピー元のアドレスを指している (b)．そこで，そのようなポインタを to-space 上のコピー先のアドレスを指すように修正する (c)．以上の動作により，ポインタの修正を含めた to-space への移動が完了する．そして最後に from-space 全体を解放することで，ルート集合から辿ることができなかったオブジェクトを全て破棄した後，to-space と from-space を入れ替える (d)．このように，Copying では GC を実行する度，二つのヒープ領域の役割を入れ替えながら動作する．

このアルゴリズムのメリットとして，オブジェクトをコピーする際に to-space の先頭アドレスから隙間無くオブジェクトを詰めていくため，GC を実行して不要なオブジェクトを回収するのと同時に，フラグメンテーションも解消できる点が挙げられる．このようなオブジェクトの再配置によってフラグメンテーションを解消する処理をコンパクションと呼ぶ．また，Mark & Sweep のスイープフェーズではヒープ全域を走査してオブジェクトがマーク済みかどうかを確認する必要があったが，Copying では from-space 全体を解放するだけでよいため，解放処理を高速に行える．このことから，

Copying は Mark & Sweep と比較してスループットに優れている．しかし，生きているオブジェクトが多い場合，オブジェクトのコピーに要するオーバヘッドが増大してしまう．また，ヒープ領域を半分しか活用できないというデメリットがある．

2.1.4 Reference Counting

Reference Counting は,「どこからも参照されなくなった時点でそのオブジェクトはゴミとなる」という考え方に基づいたアルゴリズムである．そこでこのアルゴリズムでは，他のオブジェクトからの被参照数をカウントする参照カウンタをオブジェクト毎に設ける．この参照カウンタの値は，オブジェクトの生成時やポインタの更新時に増減される．そして，ポインタの更新によって参照カウンタの値が 0 になったオブジェクト，つまりどこからも参照されていない状態になったオブジェクトは，自らをゴミと判断し自身に割り当てられたメモリ領域を解放する．

このアルゴリズムのメリットとして，各オブジェクトはそれぞれ自身のカウンタを持つため自らがゴミであるかどうかを判断可能であり，ゴミとなると即座に自身に割り当てられていたメモリ領域を解放できる点が挙げられる．つまり，他のアルゴリズムでは GC を実行するまで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解放することはできないが，Reference Counting では不要となった領域は即座に解放されるため，ゴミによってメモリが圧迫されることがない．また先述したように，Mark & Sweep と Copying には生きているオブジェクトを探索する処理が存在するが，この処理はアプリケーションプロセスを停止させてから行う必要があるため，システムの停止時間悪化に繋がってしまう．これは，オブジェクトの探索中にアプリケーションを実行した場合，各オブジェクト間の参照関係が変わってしまう可能性があり，生きているオブジェクトを正しく探索できない可能性があるためである．しかし Reference Counting の場合，オブジェクトがゴミとなる度にそのオブジェクトのみを回収するため，アプリケーションプロセスを停止してオブジェクトを探索する必要がなく，システムの停止時間を非常に短く抑えることができる．

一方，このアルゴリズムのデメリットとしては，まずスループットの悪化が挙げられる．通常，ポインタの書き換えは頻繁に発生するため，その度にカウンタ値の増減処理を行うことはスループットを悪化させる要因となってしまう．また，カウンタは最悪の場合，ヒープ領域上の全てのオブジェクトからの参照をカウントしておかなければならない．そうした状況に対応するには参照カウンタに多くのビット幅が必要となる．そのため，全てのオブジェクトに対してこの様なカウンタを用意することはメモリ領域の使用効率を大きく悪化させてしまう．また他にも，カウンタ値の更新時に

はカウンタに対する操作を排他的に実行する必要があるため，並列処理との相性が非常に悪いことや，Mark & Sweep と同様にフラグメンテーションが発生し得ることなど，様々な問題が存在する．

2.2 既存の改良アルゴリズム

前節で述べたように，現在研究されている GC アルゴリズムは全て，先述した三つの代表的アルゴリズムの組み合わせ，もしくはその改良である．特に Mark & Sweep と Copying は，現在多くの GC アルゴリズムのベースとなっており，サーバサイド Java 環境や DalvikVM などでも広く利用されている．しかし先述したように，これらのアルゴリズムはオブジェクトの探索時にアプリケーションを停止させる必要があるため，これに伴うシステムのレスポンス低下が問題となっている．本節では，このレスポンス低下を緩和するための改良アルゴリズムのうち，特に多くの実行環境で採用されている **Concurrent GC**[8] と **Generational GC**[9] を取り上げ，これらの動作と特徴についてそれぞれ述べる．

2.2.1 Concurrent GC

Concurrent GC とは，GC の処理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 GC アルゴリズムである．しかし，Concurrent GC では GC がアプリケーションとは別のスレッドで並行に動作しているため，これらのスレッド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．また，GC 実行時にコンパクションを行うことができず，フラグメンテーションが発生してしまう可能性が存在する．これは，コンパクションを行うとオブジェクトのアドレスが変化してしまい，動作中のアプリケーションの実行に支障を来すためである．つまり，フラグメンテーションを解消するためには，まずアプリケーション全体を停止させてからコンパクションを行う必要があり，その間は Concurrent GC が目的とする GC とアプリケーションの並行動作を実現できなくなってしまう．

また一般に Concurrent GC では，実際は全ての処理をアプリケーションと並行に実行することはできない．ここで，Oracle が提供する JavaVM である **HotspotVM**[10] に実装されている，Mostly-concurrent Mark & Sweep[11] というアルゴリズムを例に，Concurrent GC が抱える問題について述べる．Mostly-concurrent Mark & Sweep は Mark & Sweep を基本とするアルゴリズムである．Mark & Sweep の様な，オブジェクトにマークを付けて，そのオブジェクトの生死を管理するアルゴリズムでは，GC と

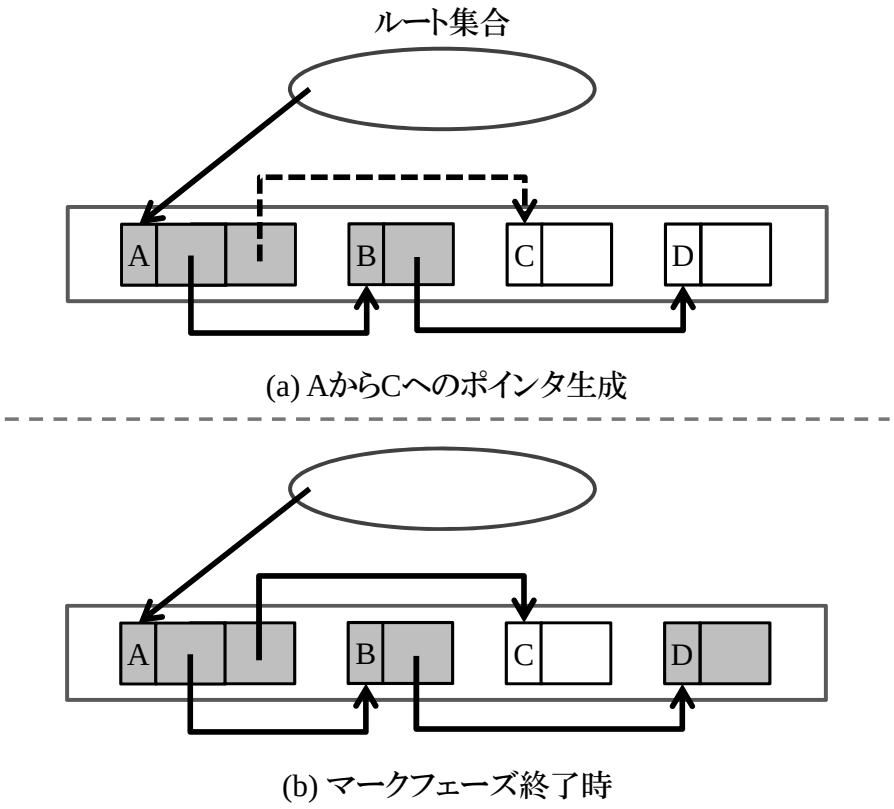

図 4: ポインタの書き換えによるマーク漏れ

アプリケーションが並行に動作すると，生きているオブジェクトへのマーク漏れという致命的な問題が発生する可能性が存在する．**図 4** にこの問題が発生する場合の動作例を示す．GC がマークフェーズを実行中，(a) のようにオブジェクト B までマークが完了した時点で，アプリケーションによってマーク済みオブジェクト A から新たにオブジェクト C へのポインタが生成されたとする．この時，GC はこのポインタの生成を検知できないため，オブジェクト C をマークしないままマークフェーズを終了してしまう (b)．そのため，オブジェクト C はその後のスイープフェーズでゴミとして扱われ，本来破棄してはいけないオブジェクトであるにも関わらず，これに割り当てられているメモリ領域を解放してしまう．

このようなマーク漏れによる誤った解放を防ぐために，Mostly-concurrent Mark & Sweep では，マークフェーズを更にいくつかのフェーズに分割し，アプリケーションと並行動作させる処理を限定することで対処している．この分割したフェーズを**図 5** に示す．Initial Mark は，ルート集合から直接参照されるオブジェクトへのマークを行うフェーズである．このフェーズは，ルート集合に含まれるポインタのアプリケーション

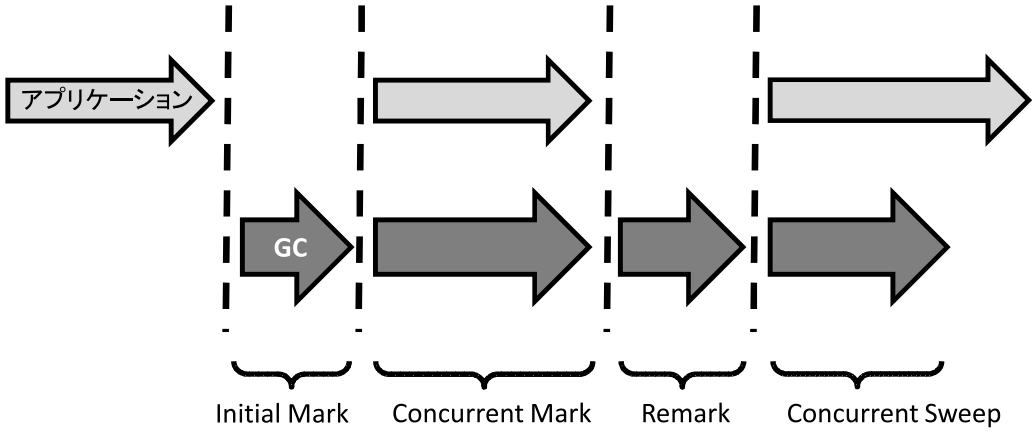

図 5: Concurrent GC における四つのフェーズ

による書き換えを防ぐため，アプリケーションを停止させて実行する必要がある．しかし一般に，ルート集合から参照されるオブジェクトは限られるため，その停止時間は短くて済む．そしてその後，Concurrent Mark フェーズで，Initial Mark フェーズでマークされたオブジェクトからポインタを辿ることのできるオブジェクトへのマークを行う．Concurrent Mark フェーズはアプリケーションの動作と並行して実行されるため，先述したような GC が検知できないポインタの書き換えが発生する可能性がある．そこで，アプリケーションはポインタを書き換えたオブジェクトに，Mark & Sweep によるマークとは別に，書き換えられたことを示すマークを行う．これは，オブジェクトのヘッダに Mark & Sweep 用とは別のマークビットを定義し，それをアプリケーションがセットすることで実現する．そして最後にもう一度アプリケーションを停止させ，Remark フェーズに移行する．このフェーズでは Concurrent Mark フェーズで書き換えられたことを示すマークが付いているオブジェクトを探索し，そのオブジェクトから再びマーク処理をやり直す．Mostly-concurrent Mark & Sweep ではこのように，Concurrent Mark フェーズでポインタの書き換えを検知し，Remark フェーズでマーク処理をやり直すことでマーク漏れによるメモリ領域の誤解放を防いでいる．なお Mostly-concurrent Mark & Sweep では，このようなアプリケーションによるポインタの書き換えをライトバリアによって検知している．しかしこのライトバリアはオーバヘッドの大きい処理であり，アプリケーションのスループットに悪影響を与えてしまう．以上で述べたように，Concurrent GC は GC 実行時の停止時間を短縮し，その発生頻度も低減させるものの，GC 実行に伴うレスポンス低下という問題の根本的解決策とは成り得ていない．

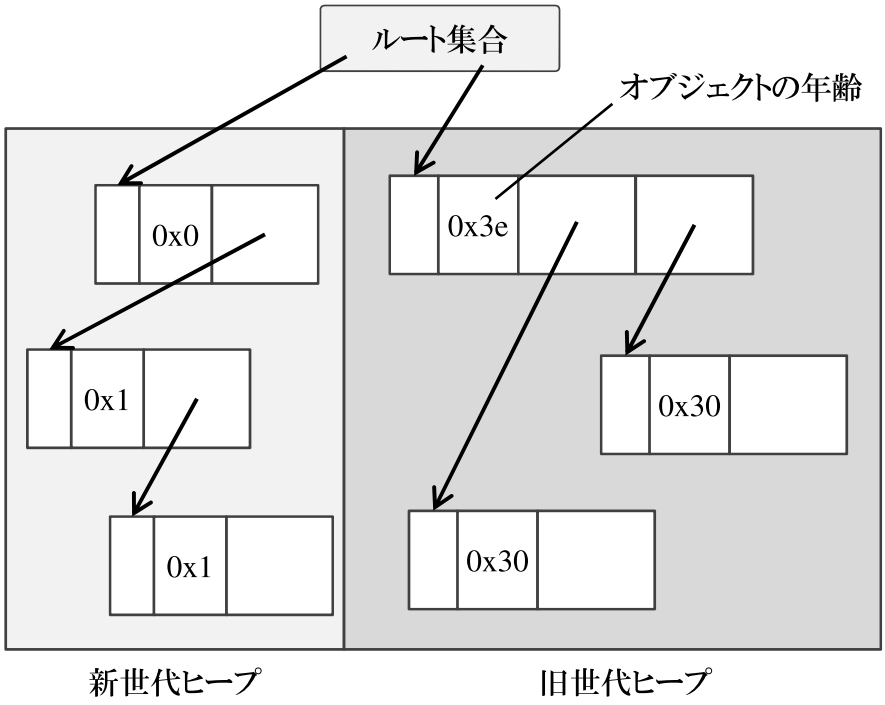

図 6: Generational GC におけるヒープ領域の分割

2.2.2 Generational GC

オブジェクトの寿命，つまりそのオブジェクトが生成されてからゴミになるまでの期間については,「多くのオブジェクトは生成されてすぐゴミとなりやすく，古くから生き残っているオブジェクトはそのまま生き続けやすい」という経験則が知られている．Generational GC はこの経験則に基づき，新しく生成されたオブジェクトと，生成後ある程度時間を経たオブジェクトを別々のヒープ領域に配置し，異なる GC アルゴリズムで管理することで GC の実行効率を向上させるアルゴリズムである．

図 6 に Generational GC におけるヒープ領域の様子を示す．Generational GC では，新しいオブジェクトが配置されるヒープ領域を新世代ヒープ，古いオブジェクトが配置されるヒープ領域を旧世代ヒープと呼ぶ．また，オブジェクトのヘッダに自身の年齢，すなわちそのオブジェクトが生成されてから実行された GC の回数を保持するフィールドが追加される．オブジェクトを生成する際は，新世代ヒープ内のメモリ領域を確保する．そして，新世代ヒープに十分な空き領域がなくなると，新世代ヒープのみを対象とした GC を実行する．この新世代ヒープのみを対象とした GC を minor GC と呼ぶ．なお，先に述べたように Copying や Mark & Sweep には，生きているオブジェクトのみを辿るという操作が存在する．そこで，多くのオブジェクトがゴミである可

能性が高い新世代ヒープ上のオブジェクトを対象とする minor GC に，それらの GC アルゴリズムを採用することで，GC にかかる時間を短縮でき，アプリケーションの停止時間を抑えることが可能となる．なお，minor GC には Copying が採用されることが一般的である．これは 2.1.3 項で述べたように，Copying が Mark & Sweep と比較してスループットに優れるためである．

一方，一定回数の minor GC を経ても回収されなかったオブジェクトは，旧世代ヒープに移され旧世代オブジェクトとして扱われる．この様に，新世代から旧世代へとオブジェクトの世代を変更することを昇格と呼ぶ．そして，minor GC の実行とそれに伴う昇格が繰り返され，旧世代ヒープが一杯になると，今度はヒープ領域全体を対象とする GC が実行される．このヒープ領域全体を対象とした GC を major GC と呼ぶ．major GC は旧世代ヒープが一杯になるまでは実行されないため，minor GC と比較するとその発生頻度は低いものの，ヒープ領域全体を走査しなければならないため，処理に時間を要してしまう．

この様に，Generational GC はオブジェクトを世代ごとに分けて管理し，それぞれに適した GC アルゴリズムを適用することでシステムのレスポンス低下を抑制しているが，major GC の発生時にはレスポンスが低下してしまうことには変わりない．そのため Generational GC を利用する際には，major GC の発生を抑制するために，オブジェクトを昇格させるための年齢の閾値や，新世代ヒープ，旧世代ヒープそれぞれの領域サイズなど様々なパラメータをユーザ自身がチューニングする必要がある．しかし，こうしたパラメータの最適値はシステムの構成や，実行するアプリケーションによって変化するため，その作業は非常に煩雑であり，ユーザの大きな負担となってしまう．更に，オブジェクトのヘッダにオブジェクトの年齢などの情報を記憶しておくフィールドを確保しなければならずヒープ領域の使用効率が低下してしまう．また Generational GC は上述したように，多くのオブジェクトは生成されてすぐゴミとなるという経験則に基づくアルゴリズムであるため，そもそもその経験則に反した，多くのオブジェクトが長期間生存するようなプログラムでは，major GC が頻発してしまいシステムの性能低下に繋がる可能性も存在する．こうした問題から，Generational GC もまた，GC 実行に伴うレスポンス低下という問題の根本的解決策とは成り得ていない．

2.3 既存のハードウェア支援手法

前節で述べた Concurrent GC や Generational GC のように，2.1 節で述べた三つの代表的アルゴリズムを基本とする GC の高速化に関する研究は，そのほとんどがソフトウェア面における改良である．しかし，わずかながらハードウェア支援による GC の高速化に関する既存研究も存在している．本節ではその中から二つの既存研究を取り上げ，それらがどのようにハードウェア支援による高速化を実現しているのかについて述べる．

2.3.1 SILENT

SILENT[12] は，NUE プロジェクト [13] によって開発された Lisp マシンである．Lisp マシンとは，Lisp で記述されたプログラムを効率よく実行するために最適化された計算機である．SILENT では GC アルゴリズムに Mark & Sweep をベースとする Concurrent GC を採用しており，GC プロセスとアプリケーションプロセスが並行して動作する．そのため 2.2.1 項で述べたような，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．

この問題に対処するために SILENT では，アプリケーションによるポインタの書き換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．まずアプリケーションは，マーク済みのオブジェクトが持つポインタを未マークオブジェクトを参照するように書き換えたとき，その参照元オブジェクトを指すポインタを，再マークを専門に行う GC プロセスが管理している専用のキューに追加する．この GC プロセスはオブジェクトへのポインタがこのキューに入れられる度，そのオブジェクトから再度マーク処理を行うことでポインタの書き換えによるマーク漏れを防ぐ．

しかし，このライトバリアはポインタの書き換えがある度に実行されるので，これをソフトウェアで実現するとプログラムの実行時間に占めるライトバリアの割合が大きくなってしまう．そこで 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，ハードウェアサポートによる高速化を実現している．なおマイクロプログラムとは，機械語よりも詳細に CPU 内部の動作を記述できるコードで書かれたプログラムのことであり，通常は ROM に格納される．このコードはゲートやフリップフロップ単位での制御が可能であるため，実行速度の面で最適化された動作を記述できる．

この高速なライトバリアによって，SILENT における GC による停止時間は最大で 100 マイクロ秒以下と非常に短時間に抑えられている．しかしこの高速なライトバリ

アは，SILENT のようにマイクロプログラム制御方式を採用しているプロセッサにおいてのみ実現可能な手法である．この制御方式は主に CISC 型のプロセッサで用いられており，組み込みシステムで広く用いられている ARM アーキテクチャ等，RISC 型のプロセッサで実現することは難しい．また，SILENT は実時間システムの実現を目的として作られた特殊なアーキテクチャ構成となっており，SILENT が用いている GC アルゴリズムは，この特殊なアーキテクチャに対して最適化されたものである．そのため，これを汎用的な計算機上で実装することは困難である．

2.3.2 Network Attached Processing

Network Attached Processing（NAP）[14] 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 Pauseless GC という，Mark & Sweep と Copying を組み合わせた GC アルゴリズムが採用されている．Pauseless GC は，生きているオブジェクトへのマークを行うマークフェーズと，マーク完了後のオブジェクトを移動させるリロケーションフェーズ，および移動対象のオブジェクトを指しているポインタを更新するリマップフェーズの三つのフェーズからなる．そして，NAP ではこれら三つのフェーズが全て，アプリケーションの実行を担う Java スレッドと並行して動作する．

仮にマークフェーズをアプリケーションと並行して動作させた場合，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，リードバリアを活用することでそのようなマーク漏れを防いでいる．NAP の場合，GC を実行するスレッドはオブジェクトを巡回する際，そのオブジェクトにマークすると同時に，そのポインタにも GC スレッドによって巡回済みであることを示すマークを施しておく．なお，Java ではアライメント制約からオブジェクトが 8 バイトの倍数値のアドレスに配置される．そのため，アドレスの値であるポインタをビット列で表すと下位 3 ビットが常に 0 となる．このことを利用し，そのうちの 1 ビットをマーク用ビットとして用いている．これを Not Marked Through（NMT）-bit と呼ぶ．そして，GC を実行するスレッドがマークフェーズを実行中，アプリケーションはポインタが参照しているオブジェクトをロードする度に，リードバリアによって当該ポインタの NMT-bit をチェックする．もし GC を実行するスレッドが未巡回であれば，NMT-bit を立てた後，GC を実行するスレッドが持つ専用のキューにこのポインタを追加することで，これを再度 GC プロセスが巡回できるようにする．このようにして NAP では，GC スレッドが未巡回のポインタをリードバリアによって検知することで，並行動作に起因するマーク漏れを防いでいる．このリードバリアは，NMT-bit 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．NMT-bit のチェックに要するオーバヘッドは通常のロード命令と比較して 1 サイクル多い程度であるため，高速にリードバリアを実行できる．

また，生きているオブジェクトへのマーク完了後は，オブジェクトの移動処理であるリロケーションフェーズ，およびポインタの更新処理であるリマップフェーズが実行される．なお，NAP ではヒープ領域をページ単位で区切っている．そこでオブジェクトの移動の際には，ゴミとなるオブジェクトのみが含まれるページを探し，まずはそのページを解放する．しかし，そのようなページは一般的に多くは存在しない．そこで次に，生きているオブジェクトが，ある一定数より少ないページを選び，十分な空き領域がある他のページへ生きているオブジェクトのみを移動させた後，移動元のページを解放する．この処理ではオブジェクトの移動を伴うため，移動元のオブジェクトを指すポインタは全て，移動先のオブジェクトを指すように更新しなければならない．そのため，通常これらの処理をアプリケーションと並行して実行することは非常に難しい．そこで NAP では，ここでもリードバリアを用いるハードウェア支援によってこれを実現している．

NAP では，リロケーションフェーズ終了後，即座にポインタの更新を行わず，移動元と移動先アドレスの対応表を作成する．そして，GC スレッドは GC-protect という処理によって移動元のページを保護する．その後，GC-protect によって保護されたページ内のオブジェクトがロードされた時，初めてそのオブジェクトを参照するポインタを修正する．このように，NAP ではリードバリアによって移動元ページに対する読み出しを検知することで，アプリケーションと並行動作可能なポインタの修正を実現している．なお GC-protect は，TLB を拡張し，OS が user-mode と kernel-mode との間に設けた GC-mode という特権レベルをサポートすることにより実現される．そして，この GC-protect によって保護されたページへのポインタがロードされると，GC-trap というトラップルーチンに処理が移され，ロードしたポインタの更新が行われる．GC を実行するスレッドはリロケーションフェーズを終えると，GC-protect による保護を解かずにマークフェーズに入る．これにより，GC を実行するスレッドがマークフェーズ実行時に，保護されたページへのポインタを全てロードすることになるので，ページ内の全てのオブジェクトのリマップが完了しページを解放できる．つまり，このアルゴリズムではリマップフェーズと次のマークフェーズが同時に進行する．

このように，NAP ではリードバリアを用いることでアプリケーションを完全に停止させることなく GC を実行することが可能となっている．しかしこのリードバリア

は，GC プロセスとアプリケーション間の同期処理にのみ必要となる処理である．そのため，リードバリアの高速化は GC の根本的な高速化と呼べるものではなく，リードバリアの発生回数に依存するものであるため，得られる効果は限定的である．また，Pauseless GC では，マークフェーズとリマップフェーズにおいてヒープ全域を走査する必要がある．そこで NAP では，豊富な CPU リソースを用いた並行処理によって，このヒープ走査にかかるコストを隠蔽している．しかし，システムの負荷が高い場合，相対的に GC 実行時のヒープ走査にかかるコストが増加し，スループットの悪化に繋がってしまう．

2.4 本研究の新規性

システムの全体性能に与える GC の影響が大きいことは，主にサーバサイド Java 環境等で古くから知られており，これまでに述べたように，GC の高速化について様々な研究がなされてきた．しかし，アルゴリズムの改良においては 2.2 節で述べたように，Concurrent GC に代表される GC の一部の処理を並行に実行することでスループットを犠牲にして停止時間を短縮するものや，Generational GC に代表されるパラメータのチューニングによってヒープ領域全体を対象とする GC の発生を抑制するものが殆どであり，GC の抱える重要な問題を根本的に解決するものではない．また，2.3 節で述べたように，GC のハードウェア支援に関する既存研究は特定言語における GC 実装で必要となる同期処理のみを高速化しているに過ぎず，その適用範囲は非常に限られている．

これに対し本研究では，代表的な GC アルゴリズムを構成する処理要素のうち，動作の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 の大幅な高速化を図る．これによって GC の実装に限定されない，多くの GC アルゴリズムの高速化を実現することが可能となる．また，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ることで，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．さらに，2.1 節で述べた三つの GC アルゴリズムをベースとする従来の研究とは異なり，本研究はハードウェア支援という新たな視点の導入により，そのような既存の GC アルゴリズムに囚われることのない，新たな GC アルゴリズムを生み出す可能性を秘めている点にも新規性がある．

3 GCの動作解析

前章で述べたように，本論文ではGCを構成する基本的な処理要素に着目し，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．そこでまず，着目すべき構成処理要素を考察するために，本章では代表的GCアルゴリズムの動作を解析し，GCのボトルネックとなっている処理を調査する．さらに，この処理がボトルネックとなっている要因を詳細に調査することで，ハードウェア支援による高速化が見込める点について考察する．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GCを対象として行う．

3.1 DalvikVMのGC実装

DalvikVMがターゲットとするモバイル機器では，ユーザ体感品質の観点から，特にGCによる停止時間が短いことが求められる．そこでDalvikVMでは，2.2.1項で述べたMostly-concurrent Mark & Sweepと同様にMark & Sweepを基本とするConcurrent GCを採用しており，マークフェーズの一部とスイープフェーズが，それぞれアプリケーションの動作と並行して実行される．

このDalvikVMのMark & Sweepでは，**Liveビットマップ**，**Markビットマップ**と呼ばれる二種類のビットマップを用いることで，効率的に不要なオブジェクトを回収している．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対応している．そして，Liveビットマップはヒープ領域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しており，それぞれ該当するオブジェクト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，オブジェクトに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．

ここで，これらのビットマップを用いて不要なオブジェクトを回収する様子を**図7**に示す．図7は，AからCの三つのオブジェクトにヒープ領域がアロケートされた後，マークフェーズにおいてAとCがマークされた場合の各ビットマップの様子を示している．ここで，GC実行時に回収すべき不要なオブジェクトとは，ヒープ領域が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を回収すれば良い．図7の場合，これに該当するオブジェ

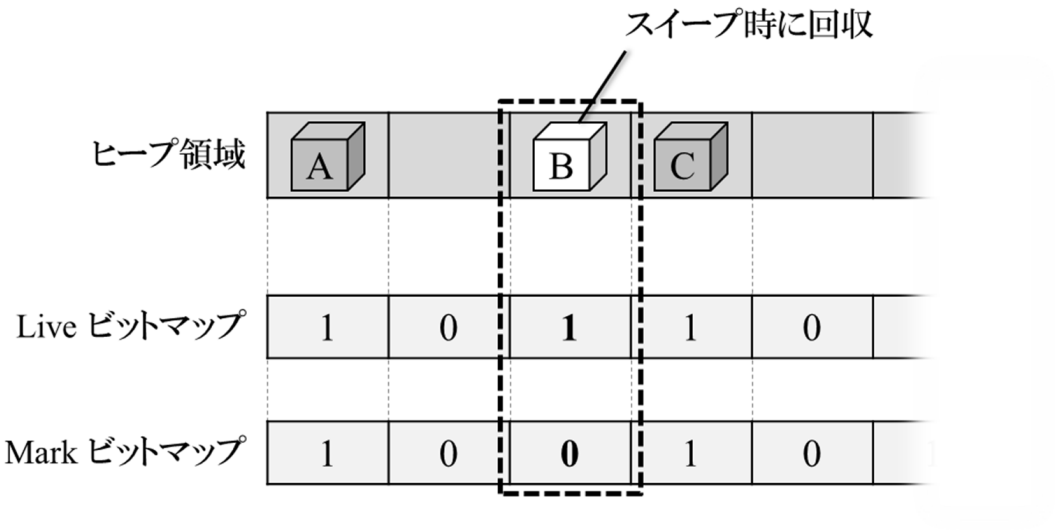

図 7: ビットマップを用いた不要なオブジェクトの回収

クトはBであり，これをスイープフェーズにおいて回収する．このように，DalvikVMでは二つのビットマップを用いた単純な演算により，不要なオブジェクトの位置を高速に特定可能である．そのため，ヒープ全域を走査して不要なオブジェクトを検索する必要がなく，スイープ処理を高速に実行できる．

なお上述したMarkビットマップのように，ビットマップを用いてマーク済みのオブジェクトを管理する手法を**ビットマップマーキング**[15]と呼ぶ．ビットマップマーキングを用いない単純なMark & Sweepの場合，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ある．つまりオブジェクトへのマーク時には，このマーク用のビットに対する操作が必要であるため，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Linuxで採用されている仮想記憶の高速化手法の一つである**コピーオンライト**[16]との相性が非常に悪い．コピーオンライトではプロセスの複製時にメモリのコピーを行わず，メモリページの書き換えが発生した時点でメモリのコピーを行う．つまりコピーオンライトには，メモリの書き換えがなければメモリコピーの必要がないという利点がある．しかし上述したような単純なMark & Sweepの場合，マーク処理に伴ってオブジェクトへの書き込みが発生するため，マーク処理を実行する度にコピーオンライトによるメモリコピーが頻発してしまい，メモリ領域が圧迫されてしまう．これは特に，メモリ容量が限られるモバイル機器では大きな問題となる．

これに対しビットマップマーキングでは，専用のビットマップを用いて各オブジェク

トへのマークを管理することで，マーク処理に伴うオブジェクトへの書き換えを防ぎ，コピーオンライトによるメモリコピーの頻発を抑制できる．しかしビットマップマーキングを用いる場合，オブジェクトへのマーク時には毎回，各オブジェクトに割り当てられているメモリアドレスの値から，ビットマップ内の対応するビット位置を計算する必要がある．そのため，オブジェクトのヘッダを利用する単純な Mark & Sweep と比較して，ビット位置の計算に時間を要する分マーク処理が遅くなってしまうという問題が存在する．

3.2 GC を構成する処理要素の内訳調査

前節で述べたように，ビットマップマーキングを採用している DalvikVM の場合，マーク処理が GC のオーバヘッドとなっている可能性がある．これを確認するために，まず DalvikVM の Mark & Sweep における各フェーズに要する実行時間の内訳を調査し，GC 処理のボトルネックについて考察した．本調査には，フルシステムシミュレータである gem5 シミュレータ [17] を用いた．そして，この gem5 シミュレータ上で DalvikVM をシミュレート実行し，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[18]，AOBench[19]，および SPECjvm2008[20] から 5 個の，計 7 個のベンチマークプログラムを用いた．

まず，Mark & Sweep の各フェーズに要する実行時間を**図 8** に示す．図 8 は，GC 処理全体の実行サイクル数に対し，マーク・スイープの各フェーズに要した実行サイクル数の割合を示しており，凡例はそれぞれ，マークフェーズに要したサイクル数（‘Mark’）とスイープフェーズに要したサイクル数（‘Sweep’），およびこれらの各フェーズの処理以外に要したサイクル数（‘misc’）を表している．

図 8 を見ると，多くのベンチマークプログラムでマークフェーズが占める割合が大きいことが分かる．特に，GCBench や crypto.signverify ではその割合が大きく，GCBench では約 80%，ctypro.signverify では約 70%にも及んでいる．なおベンチマークプログラム全体でも，マークフェーズは GC 処理全体の実行サイクル数に対し，平均約 46%の実行サイクル数を占めており，マークフェーズが GC 処理時間の大部分を占めていることが確認できた．

そこで次に，このマークフェーズにおける処理の内訳について考察する．DalvikVM のマークフェーズでは，まずルート集合から直接参照されているオブジェクトに対してマークを施した後に，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順に探索することで，生きている全てのオブジェクトに

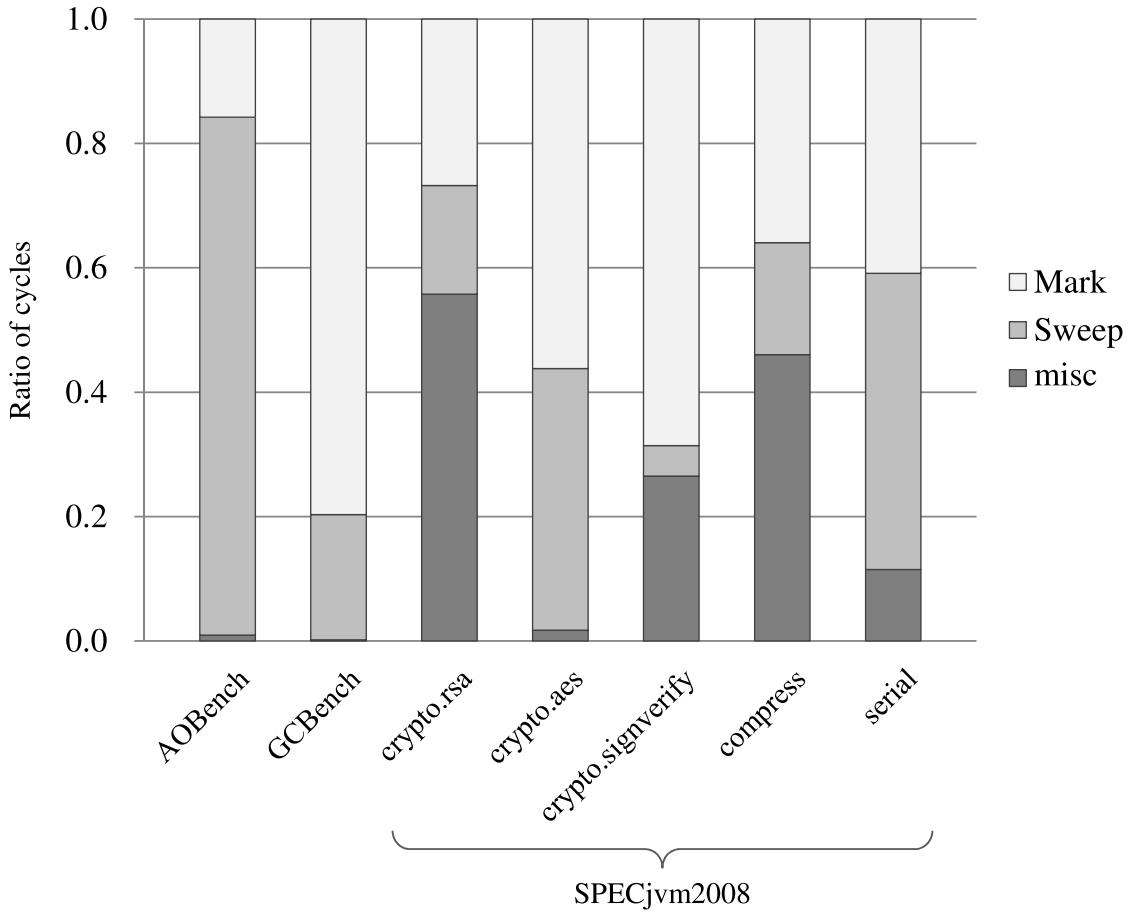

図 8: GC 実行時間に占めるマーク・スイープフェーズの割合

マークを施している．そこで，これらの各処理に要する実行サイクル数をそれぞれ調査した．その結果を**図 9** に示す．このグラフは，マークフェーズ全体に対する上記の処理の割合を示している．凡例は，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（‘MarkRoot’）と，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（‘ScanObject’）をそれぞれ示している．

結果を見ると，全てのベンチマークプログラムで ‘ScanObject’ の割合が大きいことが分かる．特に GCBench では，マークフェーズに要する実行時間のほぼ全てを ‘ScanObject’ が占めていることが分かる．これは，GCBench がツリー型のデータ構造を生成するプログラムであり，多くのオブジェクトが生成されるためであると考えられる．このような場合，それら多くのオブジェクトを探索するためのコストが大きくなってしまう．このことは，図 8 で GCBench が占めるマークフェーズの割合が大きい要因にも

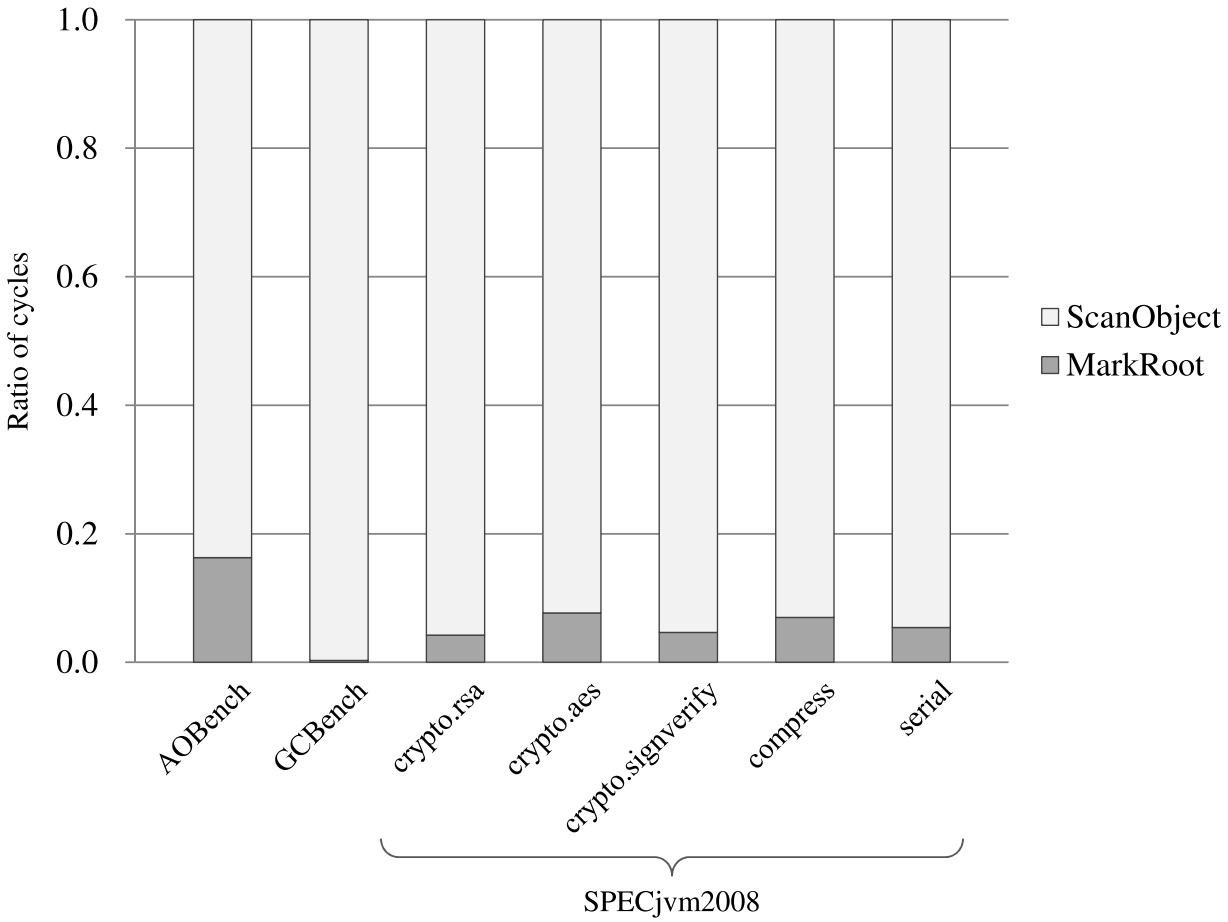

図 9: マークフェーズにおける実行時間の割合

なっていると考えられる．なお，ベンチマークプログラム全体でも，マークフェーズ全体に占める ‘ScanObject’ の占める割合は非常に大きく，平均で約 94%にも及んでいることが確認できた．

以上のことから，DalvikVM の Mark & Sweep では，実行時間の多くをマークフェーズにおけるオブジェクトの探索処理が占めていることが分かった．なお，このオブジェクトの探索処理は多くの GC アルゴリズムにおいて，GC によって回収してはならないオブジェクトを判断するために必要不可欠な処理である．例えば，2.1 節で紹介した三つの代表的アルゴリズムの内，Mark & Sweep と Copying の二つのアルゴリズムにおいて必要な処理となっている．特に Mark & Sweep は 2.1.2 項でも述べた通り，他のアルゴリズムとの組み合わせが容易であることから，実際に多くのシステムで用いられている GC アルゴリズムである．これらのことを踏まえ，本論文ではこのオブジェクトの探索処理に着目する．そして，この処理が GC 実行時間の多くを占めている要

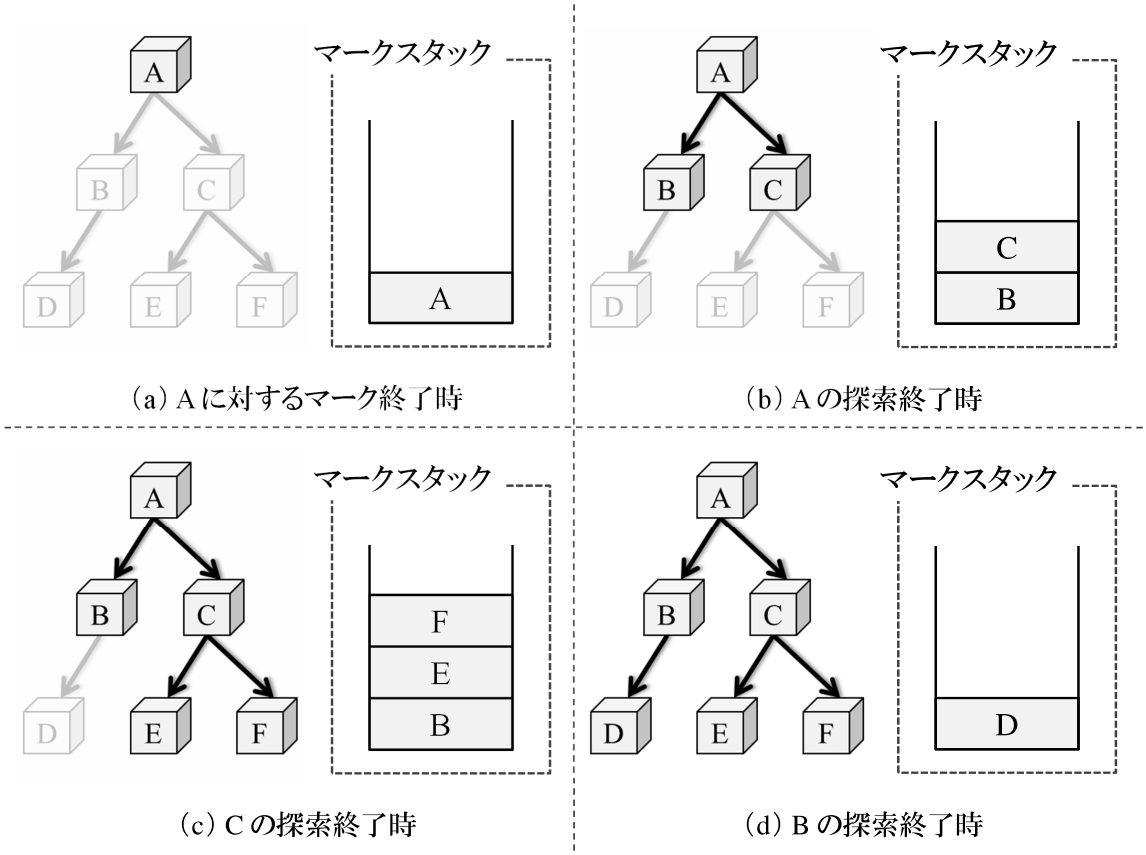

図 10: マークスタックを用いたオブジェクト探索

因を調査し，これをハードウェア支援により高速化することで，多くの GC アルゴリズムの高速化を目指す．

3.3 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．ここで，このマークスタックを用いてオブジェクトを探索する様子について，**図 10** に示す例を用いて述べる．この例は，A から F の 6 個のオブジェクト間に図に示すような参照関係が存在しており，オブジェクト A を起点としてこれらのオブジェクトを探索する場合を示している．

この例の場合，まずマークフェーズで A に対してマークを施し，さらにこれをマーク済みオブジェクトとしてマークスタックにプッシュする (a)．次にスタックから A を

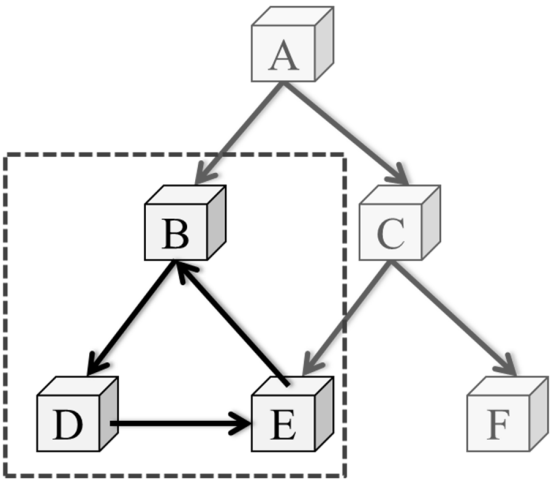

図 11: オブジェクト間に循環参照が存在する場合

ポップし，A が参照しているオブジェクトを探索する．この例では，A は B と C の二つのオブジェクトを参照しているため，これらを探索してマークを施した後，同様にマークスタックへとプッシュする (b)．A の参照を辿り終えると，その時点でスタックの先頭に積まれているオブジェクト C をポップし，C が参照しているオブジェクト E と F を探索する．そして，これらに対してマークを施すと同時にマークスタックへとプッシュする (c)．その後，同様にマークスタックからオブジェクト F と E を順にポップするが，これらは他のオブジェクトへの参照を持たないため，この時点でオブジェクト C が持つ参照の探索が終了する．次に，マークスタックから B をポップし，これが参照しているオブジェクト D をマークした後，これをスタックへプッシュする (d)．この時，オブジェクト D は他のオブジェクトへの参照を持たないため，これをマークスタックからポップした時点で，6 個全てのオブジェクトの探索が完了する．

以上で述べたように，マークスタックを用いたマーク処理では，マークを施したオブジェクトをマークスタックへ順にプッシュしていく．そして，このスタックからオブ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の場合，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．ここで，そのような循環参照が存在する例を**図 11** に示す．この例では，図中の破線で囲まれたオブジェクト B，D，E の

間に循環参照が存在している．このような場合，一度オブジェクトBをマークスタックにプッシュした後，オブジェクトEの探索時に再度Bをプッシュしてしまう．そのため，これらのオブジェクトをマークスタックにプッシュし続けてしまい，マーク処理が終了しなくなってしまう．

そこでDalvikVMでは，各オブジェクトへのマーク時に更新前のMarkビットマップを確認し，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．これにより，マーク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を防ぎ，循環参照が存在する場合にも対応可能なオブジェクトの探索処理を実現している．

しかし，このようにマークスタックへプッシュするかどうかをマーク時に判断する場合，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．つまり，過去にマークされたオブジェクトであっても，他のオブジェクトから参照されている場合には繰り返しマークしてしまう可能性がある．例えば図11の場合，オブジェクトDからEを探索してマークを施した後であっても，オブジェクトCが持つ参照を辿る際に再度Eに対してマークを施してしまう．本来，各オブジェクトへのマーク処理はオブジェクト毎に一回ずつ実行すれば良いため，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冗長な処理となってしまう．さらに3.1節で述べた通り，ビットマップマーキングを採用しているDalvikVMの場合，各オブジェクトへのマーク処理に要するコストは大きい．そのため，この冗長なマーク処理はGC処理の大きなオーバヘッドとなっている可能性が高いと考えられる．

そこで，3.2節で述べた調査に使用したものと同一のシミュレータ，およびベンチマークプログラムを用いて，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中，初めてGCが実行された際のマーク回数を全てのオブジェクト毎に測定した．そしてそのマーク回数を基にして各オブジェクトを降順にソートし，その中で上位のオブジェクト，つまり頻繁にマークされるオブジェクトに対するマーク回数の平均を求めた．その結果を**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$X = 5$の欄はマーク回数を基にソートした際の，上位5個のオブジェクトに対するマーク回数の平均を示している．

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

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	5	10	20	50	100	200
AOBench	345	254	151	83	37	20	11
GCBench	61,717	12,541	6,312	3,163	1,269	636	319
crypto.rsa	2,789	2,265	1,418	796	344	178	93
crypto.aes	2,804	1,408	854	490	218	115	61
crypto.signverify	2,801	2,238	1,412	801	349	180	94
compress	627	447	321	189	84	45	25
seial	4,269	1,929	1,156	674	313	165	87

ることが分かる．特に GCBench では，他のベンチマークプログラムと比較して非常に多くの冗長なマーク処理が発生していることが分かる．これは，3.2 節に示した評価において，GCBench 実行時の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．またベンチマークプログラム全体でも，同一のオブジェクトに対するマーク処理が数百回以上発生しているものも多く，これがマーク処理における大きなオーバヘッドの要因になっていると考えられる．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．

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の省略による 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

前章で示した調査結果より，DalvikVM では冗長なマーク処理が GC 処理の大きなオーバヘッドとなっていることが分かった．そこで本論文では，そのような冗長なマーク処理を省略することで GC を高速化する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 実行時に冗長なマーク処理を省略するためには，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し，マーク済みのオブジェクトに対するマーク処理を省略できるようにする必要がある．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，

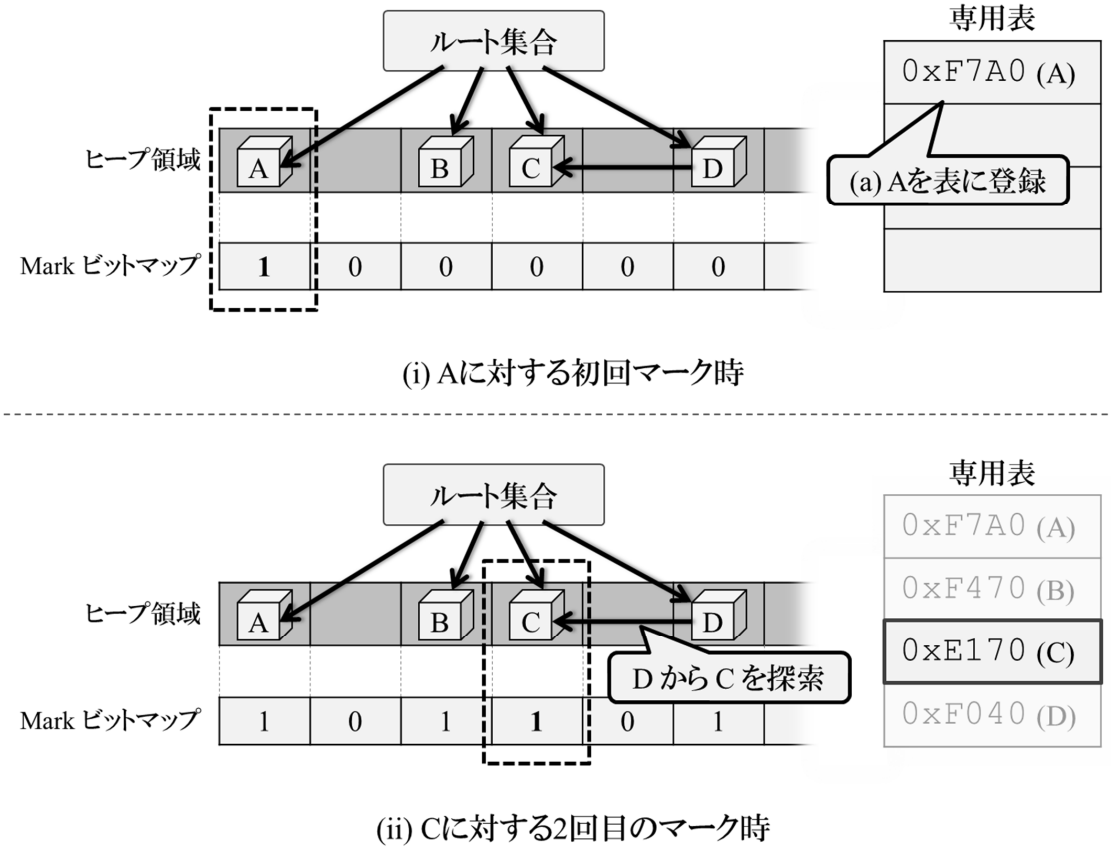

図 12: 専用表を用いたマーク処理の動作概念

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も拡張し，各オブジェクトへのマーク処理を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．以上の拡張により，各オブジェクトへのマーク処理実行前に当該オブジェクトが既にマークされているかどうかを判断可能とし，同一オブジェクトに対する冗長なマーク処理を省略する．

ここで，この専用表を用いた場合のマーク処理の動作概念を**図 12** 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場合 (i)，A はまだ専用表に登録されていないため通常のマーク処理を行う．これと同時に，提案

手法では以降の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を探索した場合，Cは既に専用表に登録されているため，このマーク処理はCに対する冗長なマーク処理であると判断できる．そこで，このC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省略する．

以上で述べたように，本提案手法では各オブジェクトへのマーク処理前に専用表を確認することで，冗長なマーク処理を省略する．これにより，既存のDalvikVMにおいて大きなオーバヘッドとなっていたオブジェクト探索処理を高速化し，GCの性能改善を目指す．

4.2　エントリの管理方法

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とで，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．しかしその場合，追加ハードウェア量が非常に大きくなってしまい，これに伴う回路面積や消費電力の増加が懸念される．特にバッテリ駆動が前提となるモバイル機器において，消費電力の増加は駆動時間の減少に伴う可用性の低下に繋がるなど，大きな問題となってしまう．

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挙げられる．つまりオブジェクトの登録時に専用表のエントリが溢れた場合，マーク対象となってから経過した時間が最も長いオブジェクトを専用表から追い出すことで，新たなエントリを確保できるようにする．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と考えられる．

ここで，そのようなLRUベースの追い出しアルゴリズムに基づくエントリの管理方法を**図13**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を順に探索する例を示している．なお，この例では説明を簡単化するために，追加する専用表のエントリ数が3であると仮定している．まずルート集合からオブジェクトAを探索し，これに対してマークを施した場合 (i)，前節で述べたように提案手法ではこれを専用表へ登録する．さらに，各オブジェクトをLRU方式で管理するためのリス

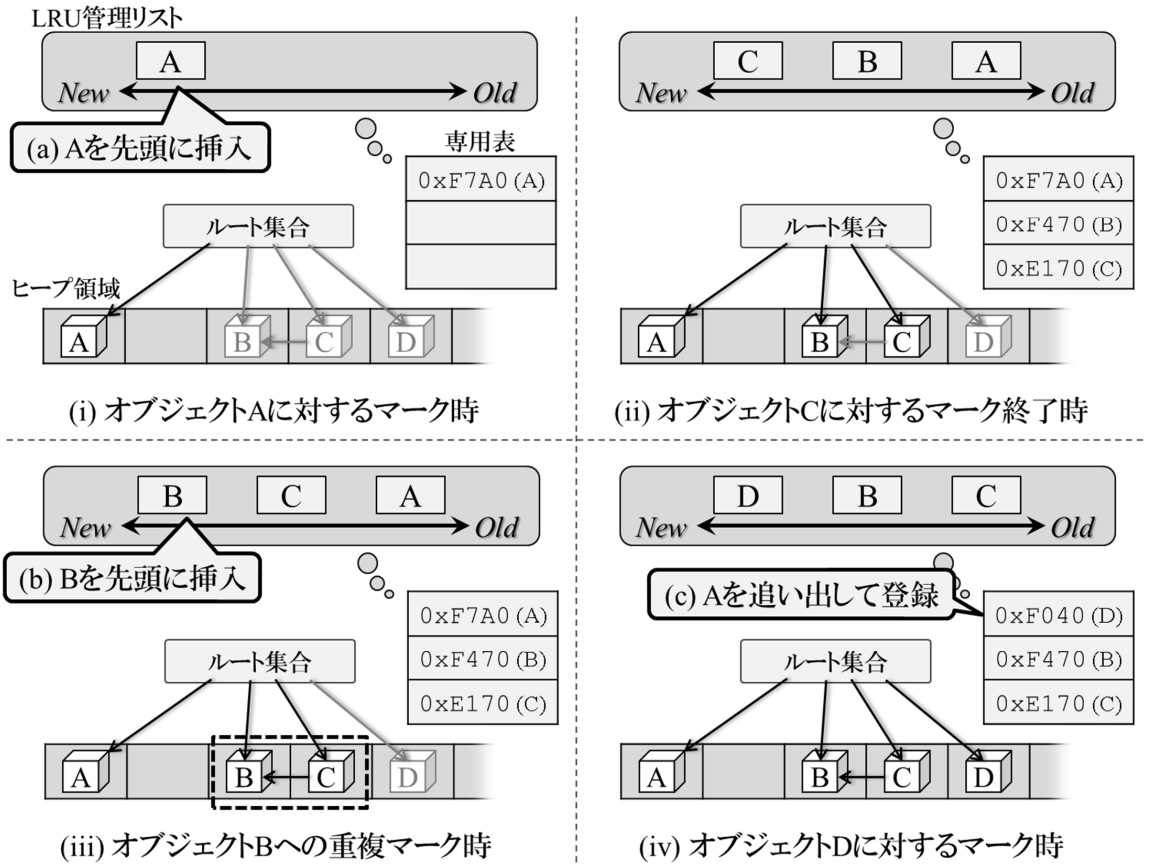

図 13: 単純な LRU 方式を用いたエントリの管理

トを用意し，その先頭にオブジェクト A に対応するエントリを挿入する (a)．この操作を，オブジェクト B と C の探索時にも同様に行うことで，(ii) に示すように専用表には A から C の三つのオブジェクトが登録され，さらにリストには先頭から順に C，B，A のオブジェクトに対応するエントリが配置される．ここで，(iii) に示すように C から B を再度探索した場合，つまり専用表に登録済みのオブジェクトを再度探索した場合も同様に，これに対応するエントリをリストの先頭へと挿入する (b)．このような動作により，マークされてから経過した時間が最も長いオブジェクトは，リストの末尾に配置されるようになる．そのため，専用表のエントリが溢れた際には，リストの末尾に配置されているオブジェクトを追い出すようにする．例えば (iv) に示すように，専用表に登録されていないオブジェクト D に対してマークを施す際には，リストの末尾に配置されていたオブジェクト A を専用表から追い出すことでエントリを確保し，そのエントリに D を新たに登録する (d)．

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，必ずしも冗長なマーク処理が頻発するオブジェクトを優先的に管理できるとは限らない．例えば，冗長なマークがなされず，マーク処理の省略が行えないオブジェクトが連続して専用表に登録された場合，それらに対応するエントリが連続してリストの先頭へ挿入されてしまうことで，今後マーク処理が頻発するオブジェクトまで追い出してしまう可能性がある．この要因として，重複してマークされているかどうかにかかわらず，全てのオブジェクトを一つの専用表を用いて統一的に管理していることが挙げられる．

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，各オブジェクトを二つの専用表を使い分けて管理する．なお本論文では，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．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．そのため，上述したようなマーク処理の省略が行えないオブジェクトが連続して専用表に登録された場合であっても，マーク処理が頻発するオブジェクトまで追い出されてしまうことがなくなる．

ここで，以上で述べたような二つの専用表を用いたエントリの管理方法を**図 14** に示す例を用いて説明する．この例では，図 13 で示したものと同様に，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場合 (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できる．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いずれかのエントリを追い出すようにす

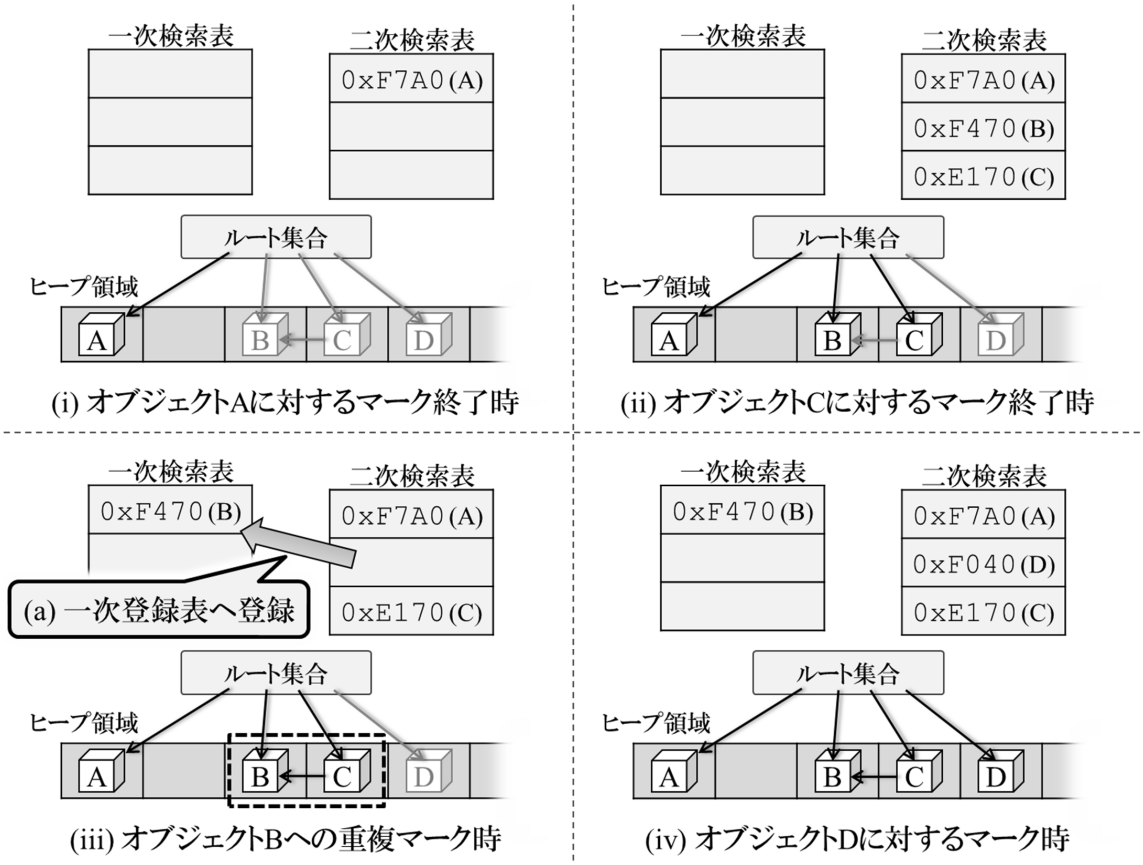

図 14: 二つの表を用いたエントリの管理

る．二次検索表のエントリを追い出すことにより，重複してマークがなされないオブジェクトが連続して表に登録される場合であっても一次検索表に登録されているオブジェクトは追い出されることがないため，マーク処理が頻発するオブジェクトを優先的に一次検索表で管理できる．

4.3 専用表を用いたマーク処理の動作モデル

これまで述べたように，提案手法では専用表を用いてマーク済みのオブジェクトを管理することで，冗長なマーク処理を省略する．これを実現するためには，各オブジェクトへのマーク処理前に専用表を確認し，当該オブジェクトがマーク済みであるかどうかを判断する必要がある．そこで，既存の DalvikVM におけるマーク処理の動作を拡張し，専用表を確認するための処理を追加する．本節では，まず既存の DalvikVM におけるマーク処理の動作を示した後，提案手法における専用表を用いたマーク処理の動作モデルについて述べる．

```
1  void scanFields(srcObj){
2    while("srcObjが参照を持つ"){
3      obj = dvmGetFieldObject(srcObj);
4      markObject(obj);
5    }
6  }
7  void markObject(obj){
8    if(obj != NULL){
9      if(setAndReturnMarkBit(obj)==0){
10       markStackPush(obj);
11     }
12   }
13 }
```

図 15: 既存の DalvikVM におけるマーク処理の簡易コード

既存の DalvikVM では探索する全てのオブジェクトに対し，Mark ビットマップ内の対応するビット位置を特定してマークを施し，さらにこれをマークスタックにプッシュするかどうか判断するために，更新前の Mark ビットを確認するという処理を毎回行っている．ここで，既存の DalvikVM におけるオブジェクトの探索，および探索したオブジェクトへのマーク処理を簡易的に表したコードを**図 15** に示す．図中の scanFields 関数（1 行目）は，引数に与えられたオブジェクト（srcObj）が持つ参照を辿るための関数であり，オブジェクト間に存在する参照は主にこの関数を用いて探索する．この関数内では，まず dvmGetFieldObject 関数を呼び出す（3 行目）．なお dvmGetFieldObject 関数は，引数に与えられたオブジェクトが参照しているオブジェクトを取得するためのものである．そして，この関数の実行によって取得したオブジェクトを引数として，オブジェクトへマークを施すための関数である markObject 関数を呼び出す（4 行目）．markObject 関数では，まず引数に与えられたオブジェクトが存在するかどうかを判定する（8 行目）．この時，オブジェクトが存在する場合には setAndReturnMarkBit 関数を呼び出し（9 行目），Mark ビットマップ内で当該オブジェクトに対応するビットをセットする．なお setAndReturnMarkBit 関数は，関数内部でマークビットをセットするだけでなく，戻り値として更新前のビットを返すようになっている．そのため，この戻り値を利用することでマークスタックへプッシュするかどうかを判断可能であり，未マークのオブジェクトの場合は markStackPush 関数を呼び出すことでこれをマークスタックへとプッシュする（10 行目）．以上の動作を，scanFields 関数の引数に与え

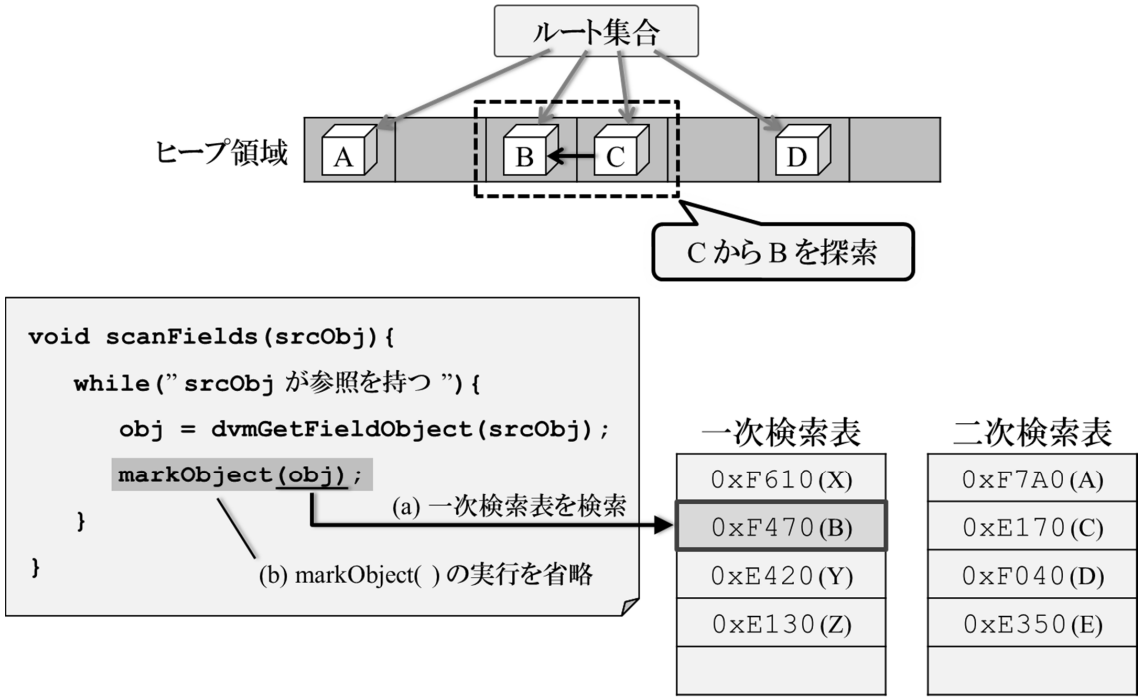

図 16: 一次検索表に登録されている場合の動作モデル

られたオブジェクトが参照している全てのオブジェクトに繰り返すことで，オブジェクト間の参照を探索している（2〜5 行目）．

これに対し提案手法では，markObject 関数を実行する際，まず一次検索表を確認するように変更する．そして表に登録済みである場合には，この markObject 関数の実行を省略し，次のオブジェクトの探索を開始することで，冗長なマーク処理を省略する．ここで，この時の動作モデルを**図 16** に示す．この図は，ヒープ領域上の 0xF470 番地に割り当てられているオブジェクト B が既に一次検索表に登録されている状態で，オブジェクト C から再度 B を探索する例を示している．提案手法では，scanFields 関数内で markObject 関数を実行する際，まず引数に与えられたオブジェクト B のアドレス 0xF470 をキーとして一次検索表を検索する (a)．この例の場合，一次検索表に対する検索がヒットするため，オブジェクト B に対するマーク処理の実行を省略し (b)，dvmGetFieldsObject 関数によって次のオブジェクトを探索する．このように，従来の markObject 関数の処理を，一次検索表を確認してから行うように変更することで冗長なマーク処理を省略する．

一方，一次検索表に登録されていない場合にはマーク対象のオブジェクトに対するマーク処理が頻発していないと判断し，通常のマーク処理を実行する．この時，提案手

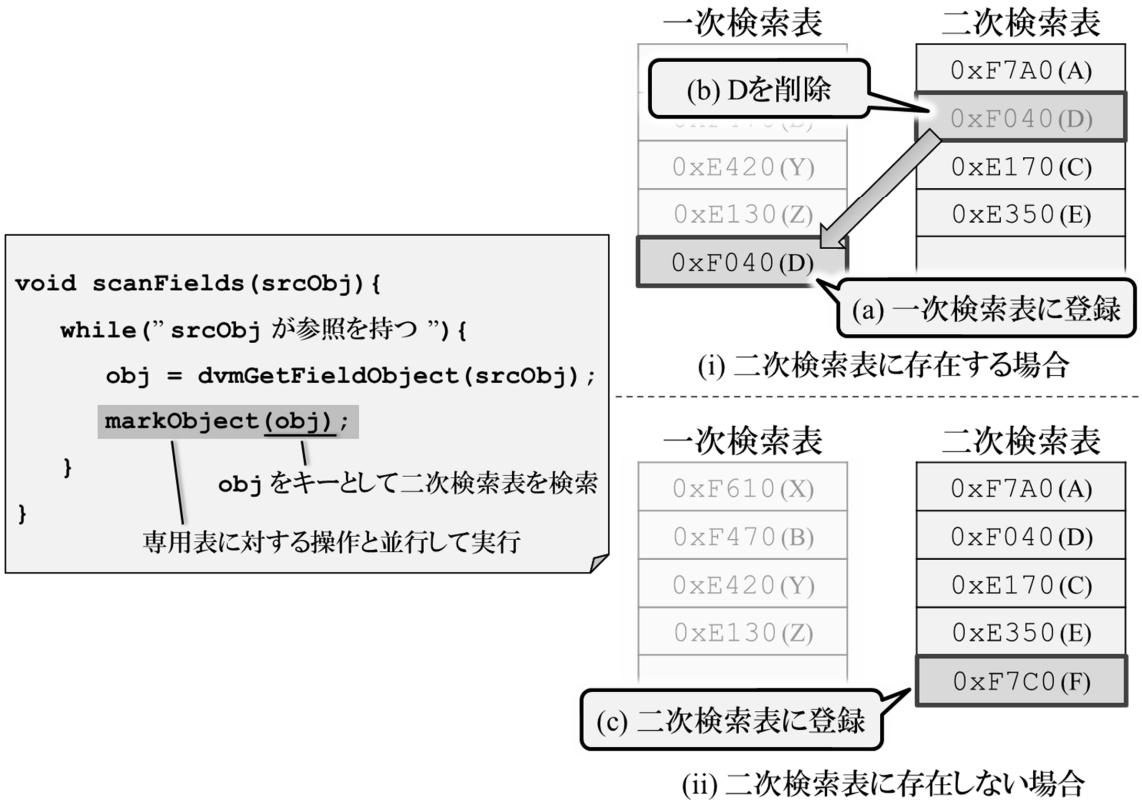

図 17: 一次検索表に登録されていない場合の動作モデル

法では当該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．ここで，この動作モデルを**図 17** に示す．二次検索表に登録されているオブジェクトをマークする場合 (i)，例えばオブジェクト D をマークする際には，これを一次検索表へ登録し (a)，二次検索表から削除する (b)．一方，二次検索表にも登録されていないオブジェクトをマークする場合 (ii)，例えばオブジェクト F をマークする際には，これを二次検索表へと新たに登録する (c)．なお，マーク処理を省略しない場合，専用表に対する操作は GC の実行に直接的には干渉しないため，これらの処理は通常のマーク処理と並行に実行できる．そこで提案手法では，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ない場合，専用表に対する処理を markObject 関数の実行と並行に行うことで，専用表の操作に伴うオーバヘッドを隠蔽する．

以上で述べたように，提案手法におけるマーク処理では，マークを行う markObject 関数の実行前に，一次検索表を検索する．そして，検索結果に応じて二つの専用表を操作し，一次検索表に登録済みのオブジェクトに対する markObject 関数の実行を省略することで，冗長なマーク処理を省略する．

5　冗長なマーク処理省略のための実装

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を実現するために必要となる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

前章で述べたように，一次検索表は過去に重複してマークがなされたオブジェクトを管理するために利用する．しかし，専用表のエントリ数は有限であるため，重複してマークがなされるオブジェクトが多く存在する場合，それら全てを一次検索表で管理できない可能性がある．そこで本提案手法では，各オブジェクトを 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リスト形式で管理する．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せるようにする．なお，前章で述べたように，専用表を用いたマーク処理では各オブジェクトへのマーク処理前に一次検索表を確認し，当該オブジェクトに対するマーク処理を省略可能かどうかを判断する．この際に必要となるエントリの検索処理を高速に行うために，一次検索表は高速な連想検索が可能な汎用 CAM（Content Addressable Memory）を用いて実装することを想定している．

ここで，以上で述べた一次検索表の具体的な構成を**図 18** に示す．なお図 18 は，一次検索表に A から D の 4 個のオブジェクトが登録されており，その中でマークされてから経過した時間が最も長いオブジェクトが D である状態を示している．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 と，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配置されているオブジェクトを記憶する 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．

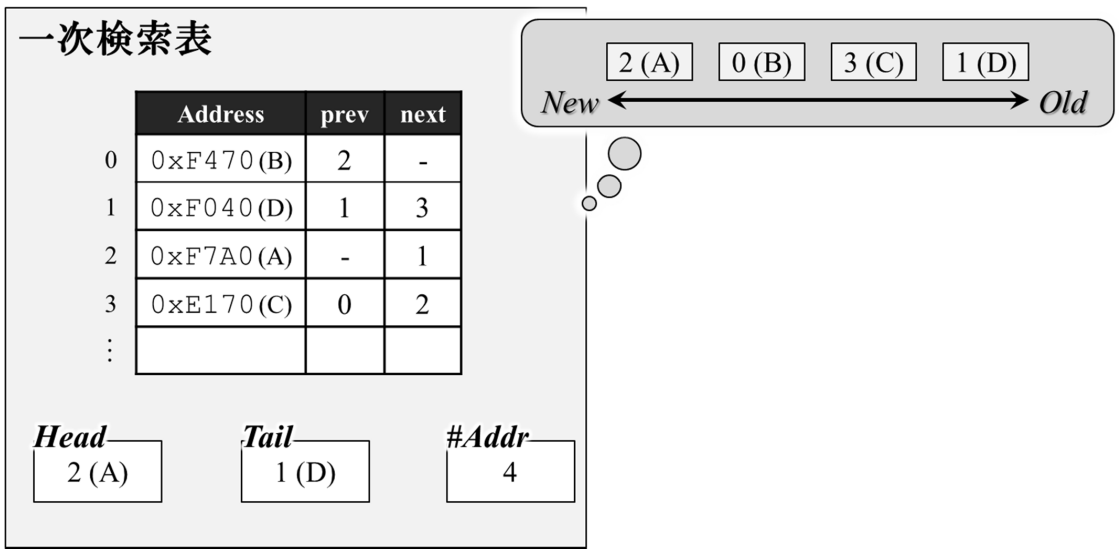

図 18: 一次検索表の構成

例えばオブジェクト B について見ると，一次検索表には B に割り当てられているヒープ領域のアドレス 0xF470 と，自身の前後に配置されているオブジェクト A と C に付与されたインデクス番号である 0 と 2 がそれぞれ登録されている．また各レジスタについて見ると，この例ではリストの先頭，および末尾のオブジェクトがそれぞれ A と D であるため，Head には A に付与されたインデクス番号である 2 が，Tail には D に付与されたインデクス番号である 1 がそれぞれ保持されている．また，#Addr には表に登録されているオブジェクト数である 4 が保持されている．このように，各オブジェクトのアドレスだけでなく，オブジェクト間の前後関係をリスト形式で記憶することで，LRU に基づくエントリの管理を可能にする．

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い場合に参照する．そして，当該オブジェクトが二次検索表に登録されている場合は，これを一次検索表へ登録する．この際に必要となるエントリの検索処理も，一次検索表と同様に CAM を用いて実装することで高速に実現できる．しかし 4.3 節で述べたように，一次検索表に対する操作とは異なり，二次検索表に対する操作は通常の GC 処理と並行して行えるため，検索コストをある程度隠蔽できる．さらに，CAM の追加は回路面積や消費電力の増加に繋がってしまうため，追加する CAM のサイズは可能な限り小容量に抑えることが望ましい．以上のことを考慮し，本提案手法では二次検索表を RAM で実装することで，CAM の追加による消費電力などの大幅な悪化を抑制する．

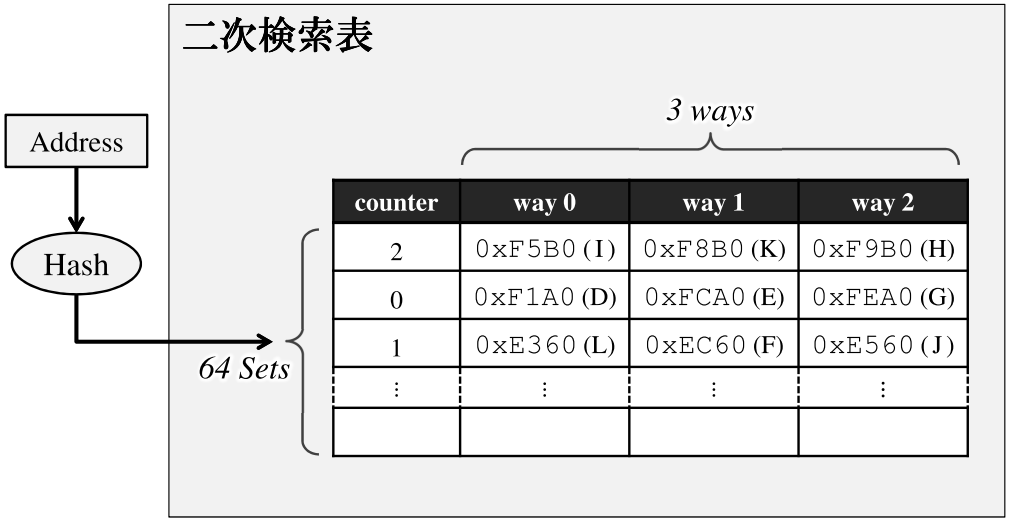

図 19: 二次検索表の構成

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，ハッシュ関数を利用することで検索コストを抑制する．

ここで，二次検索表の具体的な構成を**図 19** に示す．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を計算する．この際に利用するハッシュ関数は，オブジェクトのアドレスを入力として，ビット演算，およびシフト演算を用いることでハッシュ値を計算するものであり，求めたハッシュ値に応じてオブジェクトを登録するセットを決定する．またこれらの各セットは，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ためのフィールドをウェイ数分保持している．なお，この図はウェイ数を 3 と仮定した場合を示している．さらに，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追い出すエントリを決定するため，セット毎に独立したカウンタ値を保持するためのフィールド Counter も用意する．このカウンタは，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とし，次にオブジェクトを登録すべきウェイ番号を管理するために利用する．つまり，オブジェクトを登録する度にカウンタの値をインクリメントすることで，各ウェイに対して順にオブジェクトを登録していく．なお，対応するセット内に空きエントリがない状態でオブジェクトを登録する際には，このカウンタの値が指すウェイを上書きすることで，エントリの追い出しを実現する．例えば，図 19 においてオブジェクト L が格納さ

れているセットを見ると，このセットには三つのオブジェクトが格納されており，次に当該セットに対してオブジェクトを登録する際には，いずれかのエントリを追い出す必要がある．この時，このセットに対応するカウンタの値は 1 であるため，追い出すエントリはウェイ番号が 1 のエントリに格納されているオブジェクト F となる．

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，この時に必要となる一致比較は，最大でもウェイ数と等しい回数のみ実行すればよい．そのため，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．

このように，二次検索表のデータ構造をセットアソシアティブ構成とすることで，表の検索に要するコストを抑制する．そしてセット毎にカウンタを用意し，その値に応じてエントリを追い出すことで，多くのオブジェクトが登録された際のエントリの追い出しを実現する．

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・二次検索表のいずれかに登録されている場合と，いずれにも登録されていない場合の三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重複して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，当該オブエ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 20** 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一次検索表に対する操作例を示している．このような場合，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

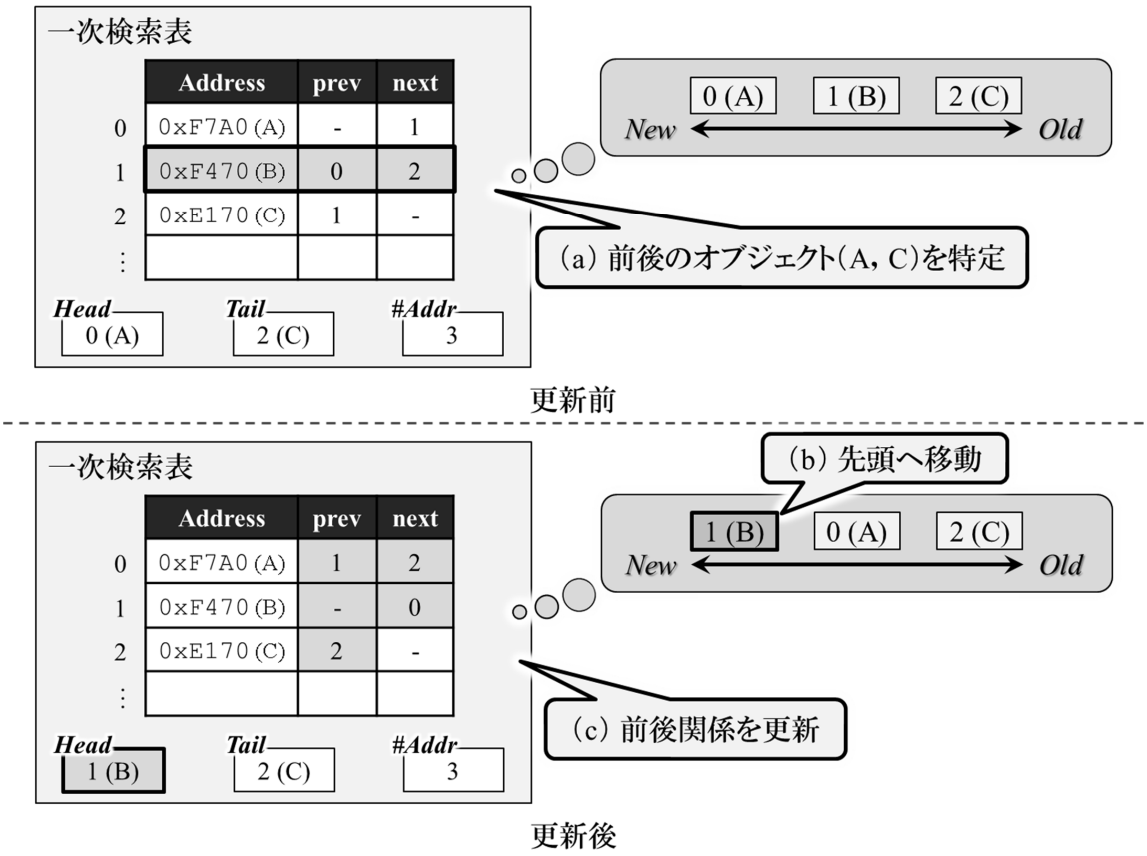

図 20: 一次検索表に登録されている場合の動作モデル

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 の次のオブジェクトを C に変更するために，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 の前のオブジェクトを A に変更するために，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．このように，オブジェクトの隣接関係を更新する際は，各エントリが保持しているインデクス番号を更新することでこれを実現する．また，この例ではリストの先頭のオブジェクトが A から B に変わったため，これに合わせて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 二次検索表に登録されている場合

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

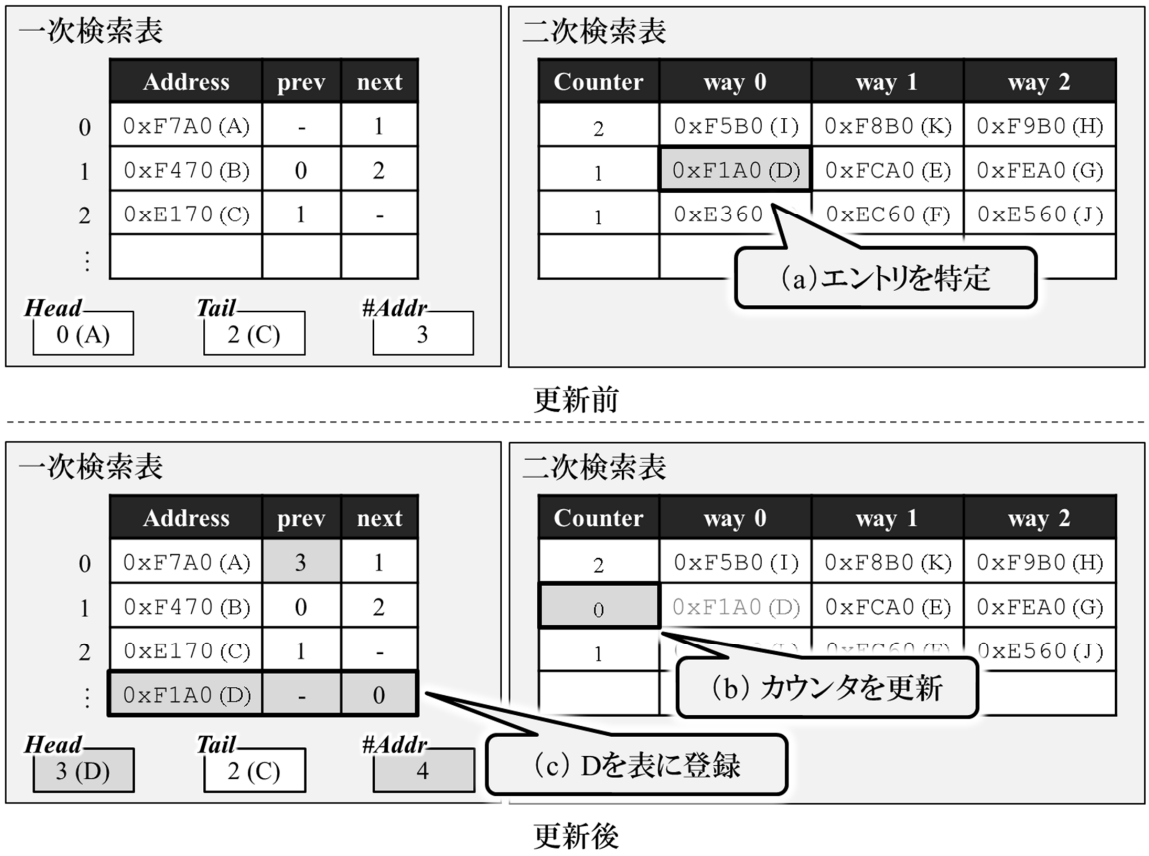

図 21: 二次検索表に登録されている場合の動作モデル

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 21** 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例を示している．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エントリを特定する (a)．次に，この D を一次検索表へ登録するために，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セットのエントリに空きが生じることになる．そこで，このエントリに対応する Counter の値を D が格納されていたウェイ番号に更新し，次のオブジェクトをこのエントリに登録するように変更する (b)．これにより，エントリの削除に伴って生じた空きエントリが存在する状態で他のエントリにオブジェクトが登録されるということがなくなるため，各セットを有効に利用できるようになる．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．この例では，更新前の一次検索表が管理するリストの先頭は A であったため，これを記憶しているレジスタ Head の値

を，D に付与されたインデクス番号 3 へと更新する．また，A が D の次のオブジェクトになるように，A の prev を D に付与されたインデクス番号である 3 に，D の next を A に付与されたインデクス番号である 0 に更新する．さらに，一次検索表が管理するオブジェクト数を示すレジスタ#Addr の値をインクリメントし，3 から 4 に更新する．

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．

5.2.3　いずれの表にも登録されていない場合

マーク対象のオブジェクトがいずれの表にも登録されていない場合，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，Counter 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，Counter の値をインクリメントする．ここで，この時の専用表に対する操作について**図 22** に示す例を用いて述べる．なお，この例ではいずれの表にも登録されていないオブジェクト X を二次検索表へ登録する際の操作を示している．

まず，X のアドレスから求めたハッシュ値を用いてこれを登録するセットを特定する．この例では，二次検索表の上から 2 番目のセットに X を登録するものとする．そして，X を登録するセットを特定した後，当該セットに対応する Counter の値を取得し (a)，その値が指すウェイに X を登録する．図 22 では Counter の値が 1 であるため，way1 に対して X を登録する (b)．なお，この登録動作はオブジェクトの登録先エントリが既に他のオブジェクトを保持しているか否かを考慮せずに行う．つまり，この例では way1 が既にオブジェクト E を保持しているが，これを X へと上書きする．これにより，あるセットに対してウェイ数以上のオブジェクトを登録しようとする場合，登録操作とエントリの追い出しを同時に実現する．そして X を登録した後，Counter の値を更新し，その値を 1 から 2 へと変更する (c)．

以上で述べたような操作により，一次検索表を用いてマーク処理が頻繁になされるオブジェクトを優先的に管理することが可能となる．そして，既存の GC 処理ルーチンを拡張し，これらの操作を追加実装することで，専用表を用いたマーク処理の省略を実現する．

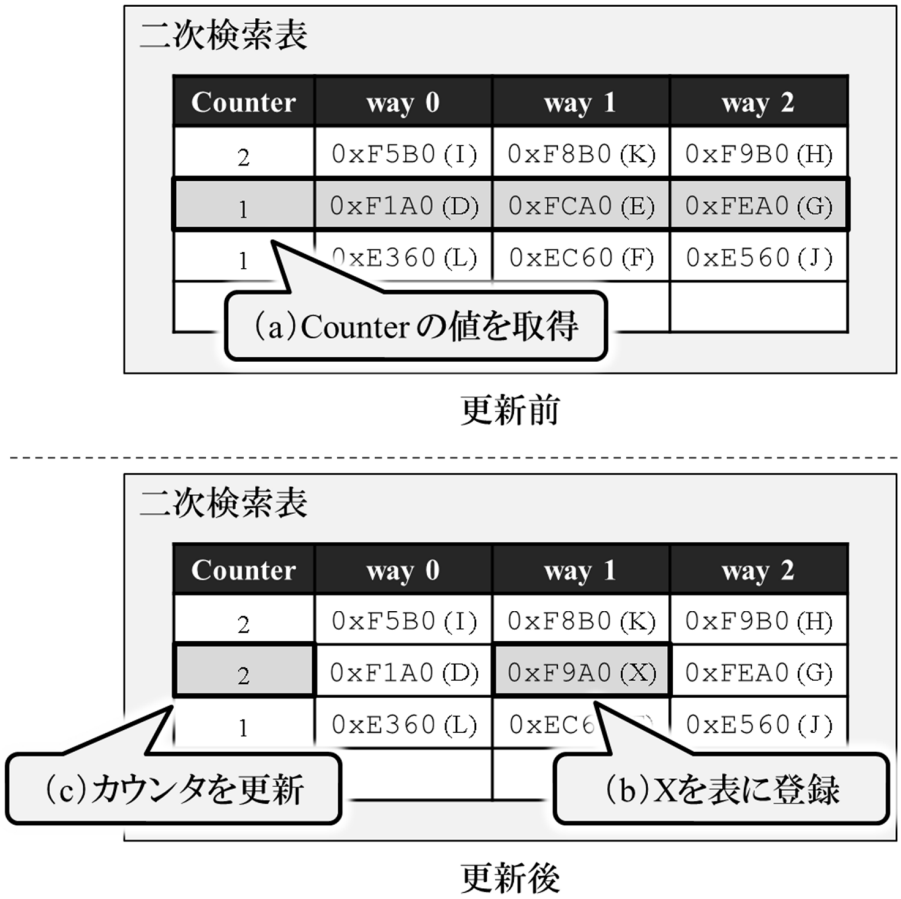

図 22: いずれの表にも登録されていない場合の動作モデル

5.3 ソフトウェアインタフェースの実装

本節では，まず前節で述べたような各専用表に対する操作を実現するための命令セット拡張について述べた後，その拡張命令をユーザが利用するためのソフトウェアインタフェースの実装について述べる．

5.3.1 表を操作する専用命令

これまでに述べたように，本提案手法では既存の GC 処理ルーチンを拡張し，専用表を操作するための処理を新たに追加する必要がある．そこで，既存の命令セットアーキテクチャを拡張し，表への操作を行う専用命令を新たに実装する．そして，その拡張命令を既存のオブジェクト探索処理を実現する命令列の適切な位置に挿入することで，提案手法における追加ハードウェアを利用できるようにする．

ここで，既存の GC 処理ルーチンへ専用命令を挿入した命令列の一例として，各オブジェクトへのマーク処理を実現する命令列に対し，専用表を参照するための専用命

```
1  cmp          r0 , #0
2  bxeq         lr          ; Nullの場合にマーク処理を終了
3  hwgc_search_table        ; 専用表を検索
4  bxeq         lr          ; マーク済みの場合にマーク処理を終了
5  mov          r1 , r0     ; 引数を指定
6  bl           42b14       ; マーク処理を実行
```

図 23: 専用命令挿入後のアセンブリコード例

令を挿入したアセンブリコードを**図 23** に示す．なお，この図に示すアセンブリ命令は DalvikVM のターゲットプラットフォームである ARM アーキテクチャのものであり，図 15 に示した markObject 関数に対応するアセンブリコードの一部を簡易的に表したものである．なお 3 行目の hwgc_search_table は，提案手法において専用表を参照するために新たに挿入した専用命令を表している．また，6 行目の bl 命令による無条件分岐先のアドレス 42b14 は，マーク処理ルーチンの先頭アドレスを表している．

まず，markObject 関数では引数 r0 に与えられたオブジェクトが NULL かどうかを判定し，これが NULL の場合，つまりマーク対象のオブジェクトが存在しない場合は markObject 関数を終了する（1，2 行目）．NULL でない場合，専用命令を実行することで当該オブジェクトが専用表に登録されているかどうか，つまり既にマーク済みであるかどうかを確認する（3 行目）．そして専用表に登録済みである場合，これに対するマーク処理を省略するために，その時点で markObject 関数を終了する（4 行目）．一方，専用表に登録済みでない場合には，引数を格納するためのレジスタである r1 に当該オブジェクトを代入し，マーク処理を実行する（5，6 行目）．

以上で述べたように，既存の命令列に専用命令を挿入し，マーク処理の本体部分を実行する前に専用表を確認するように処理を変更することで，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．なお上述したように，この専用命令は既存の命令セットアーキテクチャを拡張して実装することを想定しているが，現段階の実装では既存の命令セットアーキテクチャに含まれる nop 命令を擬似的に専用命令として代用しており，これをインラインアセンブラによってアセンブリコード内に挿入している．さらに，各命令を実行する際にプログラムカウンタの値を取得するようシミュレータを拡張し，その値と挿入した nop 命令のプログラムカウンタの値を比較することで，専用命令として使用する nop 命令を検知できるように変更した．

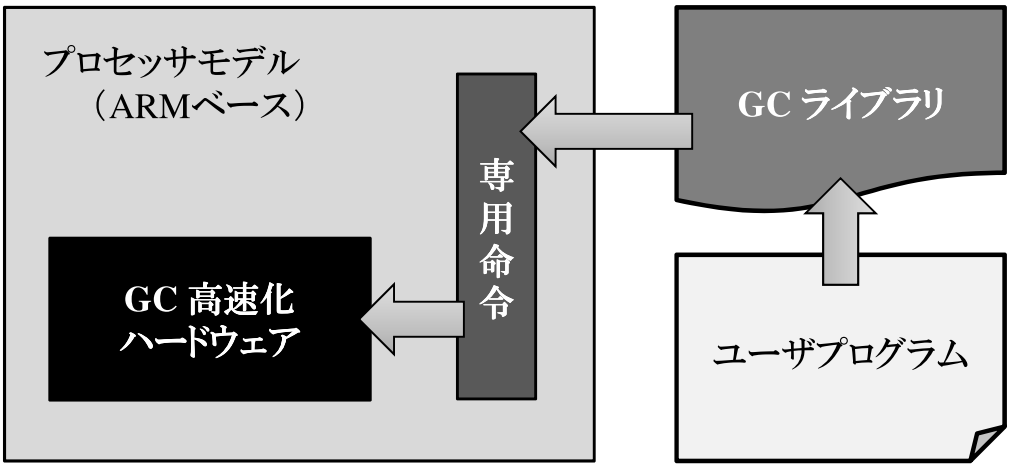

図 24: GC 高速化ハードウェアの利用

5.3.2 専用命令を利用するための GC ライブラリ

前章で述べたように，現段階の実装ではインラインアセンブラによって DalvikVM のコード中に命令を記述することで，専用命令の挿入を実現している．しかし，こうした作業は GC の実装方式とターゲットマシンのアーキテクチャに対する深い知識を必要とするものであり，一般のユーザにとっては困難である．そこで本研究では，ユーザが専用命令をプログラム中から容易に使用可能とするため，前項で示したような専用命令を追加した GC 処理ルーチンを，既存の GC 処理ルーチンと置き換え可能なライブラリ関数として提供することを想定している．ここで，ライブラリ関数を用いてユーザプログラムから GC 高速化のためのハードウェアを使用する様子を**図 24** に示す．この図に示すように，専用命令を使用する GC ライブラリを利用することで，命令セットアーキテクチャおよびハードウェアの拡張を隠蔽することができ，提案手法の高い可用性を実現できる．

ここで，4.3 節で示したオブジェクトを探索するための scanFields 関数を，専用命令を用いるように置き換えたコードの例を**図 25** に示す． この図は，4.3 節で示した既存の scanFields 関数 (a) と，これを専用表を利用するために拡張したもの (b) を示している．既存の scanFields 関数内では，まず拡張前と同様にオブジェクト間の参照を取得する（(a)，3 行目）．その後，取得した参照が指しているオブジェクトに対してマークを施すための markObject 関数を実行する（(a)，4 行目）．これに対し拡張後の scanFields 関数では，markObject 関数の実行前に専用表を確認するための GC ライブラリ関数_HWGC_isRegistered を実行する（(b)，4 行目）．これにより，専用表に登録されていないオブジェクトに対してのみ markObject 関数を実行するように変更し

```
1 void scanFields(srcObj){
2     while(...){
3         obj = dvmGetFieldObject(srcObj);
4         markObject(obj);
5     }
6 }
```

（a）既存の scanFileds 関数

```
1 void scanFields(srcObj){
2     while(...){
3         obj = dvmGetFieldObject(srcObj);
4         if(_HWGC_isRegistered(obj) != false){
5             markObject(obj);
6         }
7     }
8 }
```

（b）拡張後の scanFields 関数

図 25: GC ライブラリを用いたのコードの置き換え

表 2: シミュレータ諸元

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

（(b)，5 行目），冗長なマーク処理の実行を省略する．

このように，専用表を用いて行う処理を GC ライブラリ関数として提供することで，一般のユーザであっても簡単な記述のみで提案手法を利用できる．そのため，今後は以上で述べたような ISA の拡張，および GC ライブラリ関数の実装を進めていき，提案手法の汎用性向上についても考察していく予定である．

6 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評価し，得られた評価結果から提案手法が GC の性能に与える影響について考察する．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．プロセッサには，組み込みシステムで広く用いられている ARM アーキテクチャを選択した．ARMv7[21] は，32 ビットの RISC マ

表 3: 使用したベンチマークプログラム

プログラム	内容	ヒープサイズ
GCBench	ツリー型データ構造の作成	16 MB
AOBench	レンダリング	16 MB
SPECjvm2008		
crypto.rsa	RAS の暗号/復号化	16 MB
crypto.aes	AES の暗号/復号化	16 MB
crypto.signverify	デジタル署名/検証	16 MB
serial	シリアル通信	128 MB
compress	データ圧縮	32 MB

イクロプロセッサ，ARM-RealView PBX[22] は，ARMv7 を搭載するシステム開発用ベースボードである．

DalvikVM 上で実行するベンチマークプログラムには，GCBench，AOBench，また汎用ベンチマークプログラムである SPECjvm2008 から 5 個の，計 7 個を使用した．これらのベンチマークプログラムの一覧を**表 3** に示す．なお，DalvikVM のヒープサイズはデフォルトで 16MB であるが，このサイズで各プログラムを実行したところ，serial と compress の二つのプログラムで，メモリ不足に起因する OutOfMemory エラーが発生したため，serial には 128MB，compress には 32MB のヒープ領域を割り当てて評価を行った．

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しているため，一次検索表のサイズが大きいほど，より GC 実行サイクル数を削減できると考えられる．しかし，3.3 節の表 1 に示した調査結果を見ると，冗長なマーク処理が頻発しているオブジェクトは，マーク回数が特に多い上位数十個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．以上のことを踏まえ，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（Tlanslation Lookaside Buffer）が一般的に十数エントリから数十エントリであることを踏まえ，50 エントリとして評価を行った．また二次検索表は，頻繁にマークされるオブジェクトをあまり追い出すことなく管理できるサイズが望ましい．ここで，再度表 1 に示した調査結果を見ると，

$X = 200$ の欄，つまりマーク回数の多い上位 200 個のオブジェクトに対するマーク回数の平均は 100 未満であるプログラムが多く，上位のオブジェクトと比較すると非常に少ないことが分かる．つまり，二次検索表を 200 エントリ程度で実装すれば，その中には頻繁にマークされるオブジェクトが高い確率で登録されると考えられる．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，各ベンチマークプログラムについて，まず GC の実行サイクル数を計測し，提案手法が GC の性能に及ぼす影響を評価した．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，各ベンチマークプログラムの総実行サイクル数と，GC による平均停止時間も計測した．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，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．本節では，これらの評価結果について順に述べる．

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を**図 26** に示す．図 26 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

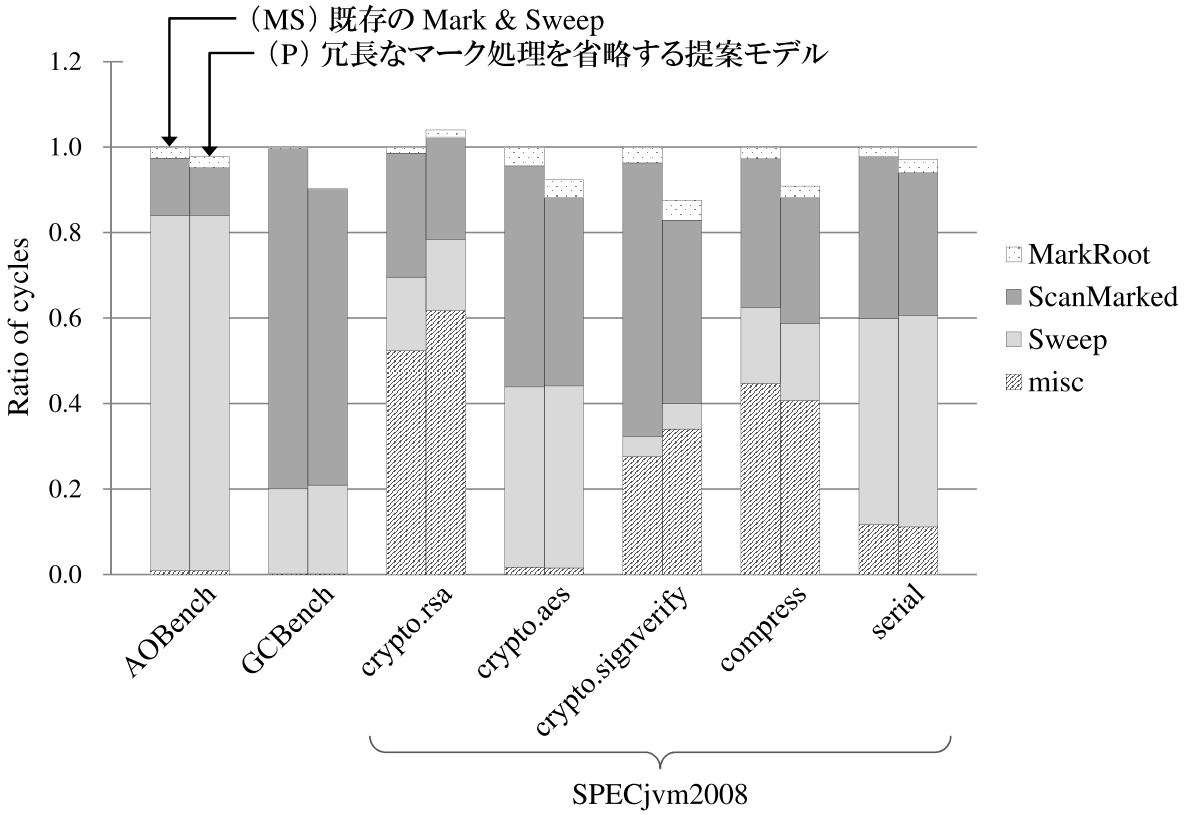

図 26: GC 実行サイクル数

マークプログラム全体でも，最大 12.4%，平均 5.7%の GC 実行サイクル数を削減できていることが確認できた．

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（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の場合，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，全体に占める GC 実行サイクル数の割合が小さいため，シミュレータ実行時のばらつきによる性能差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，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．そこでまず，一次検索表のアクセスレイテンシについて考察する．各専用表に必要なハードウェアコストの詳細は 6.4 節で後述するが，本提案手法で使用する一次検索表は 300Byte の CAM で構成可能であ

る．このサイズは，シミュレート対象マシンの TLB のサイズが 1KByte であることと比較しても十分に小さい．そこで，一次検索表はシミュレート対象マシンの TLB と同じ 2cycle で参照できると仮定する．次に，二次検索表のアクセスレイテンシについて考察する．本提案手法で使用する二次検索表は，784Byte の RAM で構成可能である．このサイズは，シミュレート対象マシンの L1 キャッシュが 64KByte であることと比較しても十分に小さいことから，二次検索表はこれと同じ 1cycle で参照できると仮定する．以上のアクセスレイテンシを，各専用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，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の比率はさらに小さくなると考えられる．

6.2.2 総実行サイクル数

次にベンチマークプログラムの総実行サイクル数の評価結果を**図 27** に示す．この図では各ベンチマークプログラムの結果を図 26 で示した二つのモデルに

(CO) 既存の Concurrent GC を実行するモデル

を加えた 3 本のグラフで実行サイクル数を示しており，既存モデル (MS) の実行サイクル数を 1 として正規化している．また凡例は，GC に要したサイクル数と，GC 以外の実行に要したサイクル数をそれぞれ示している．

結果を見ると，(CO) では一部のベンチマークプログラムにおいて総実行サイクル数が増加していることが分かる．特に AOBench では，Mark & Sweep の実行サイクル数と比較して約 1.7 倍と大きく増加してしまっている．これに対し提案モデル (P) では，いずれのベンチマークプログラムにおいても，(MS) と同程度の結果となっており，実行サイクル数の大幅な増加を防ぐことができている．

つまり提案手法を用いた場合，既存の Concurrent GC のようにスループットを悪化させることなく，GC の実行サイクル数を削減できることが確認できた．なお，このスループットの悪化は消費エネルギーの増加に繋がってしまうため，特にバッテリ駆動が前提となるモバイル機器では大きな問題となる．そのため，スループットを悪化させることなく GC 実行サイクル数を削減できる本提案手法は，モバイル機器にとって重要な問題となる消費エネルギーの増加を防ぎ，その可用性を高く保つことが可能であると期待できる．

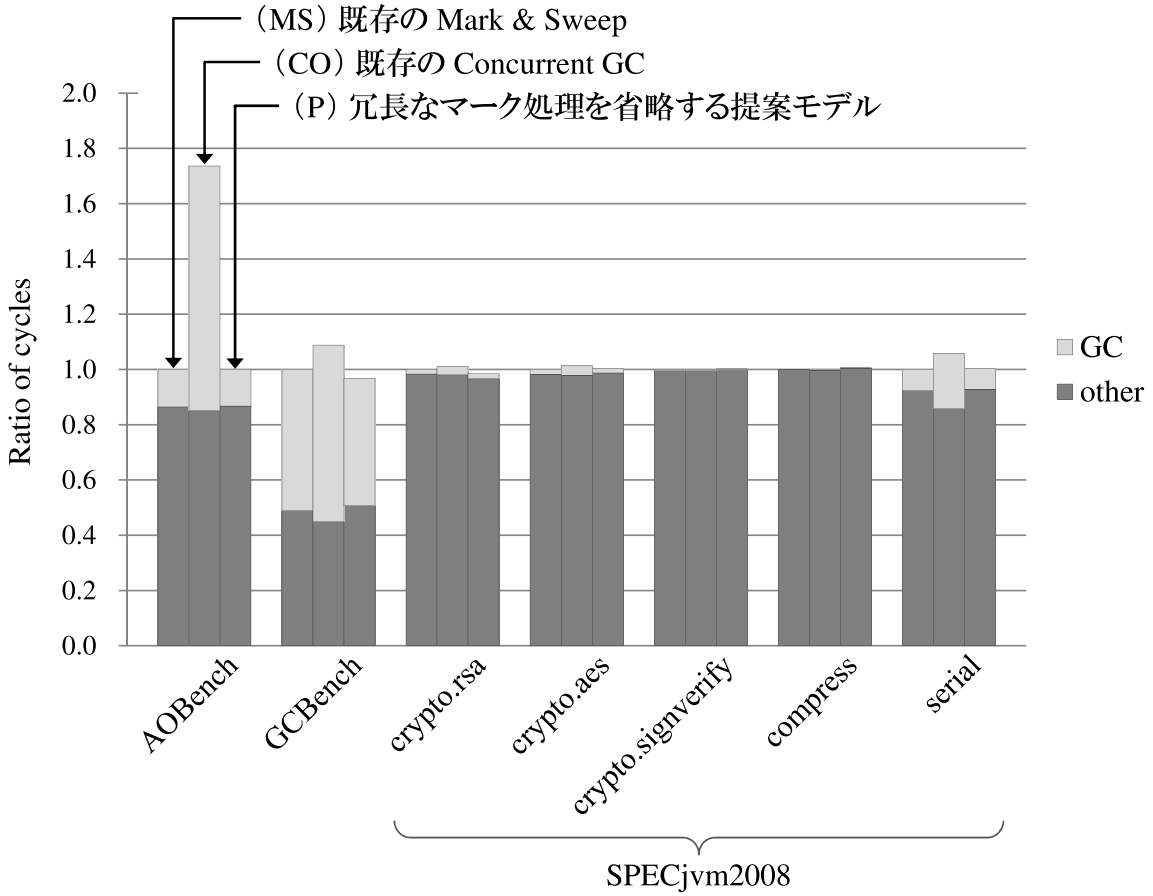

図 27: 総実行サイクル数

6.2.3 GC による平均停止時間

最後に GC による平均停止時間の評価結果を**図 28** 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，各 GC 実行時の停止時間の平均を算出した．図 28 は，図 27 と同様の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) 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．また 2.2.1 項で述べたように，Concurrent GC は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 では参考モデル (CO) で却って停止時間が大きく悪化してしまっている．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対しても，提案モデルは停止時間の悪化を防ぐことができてい

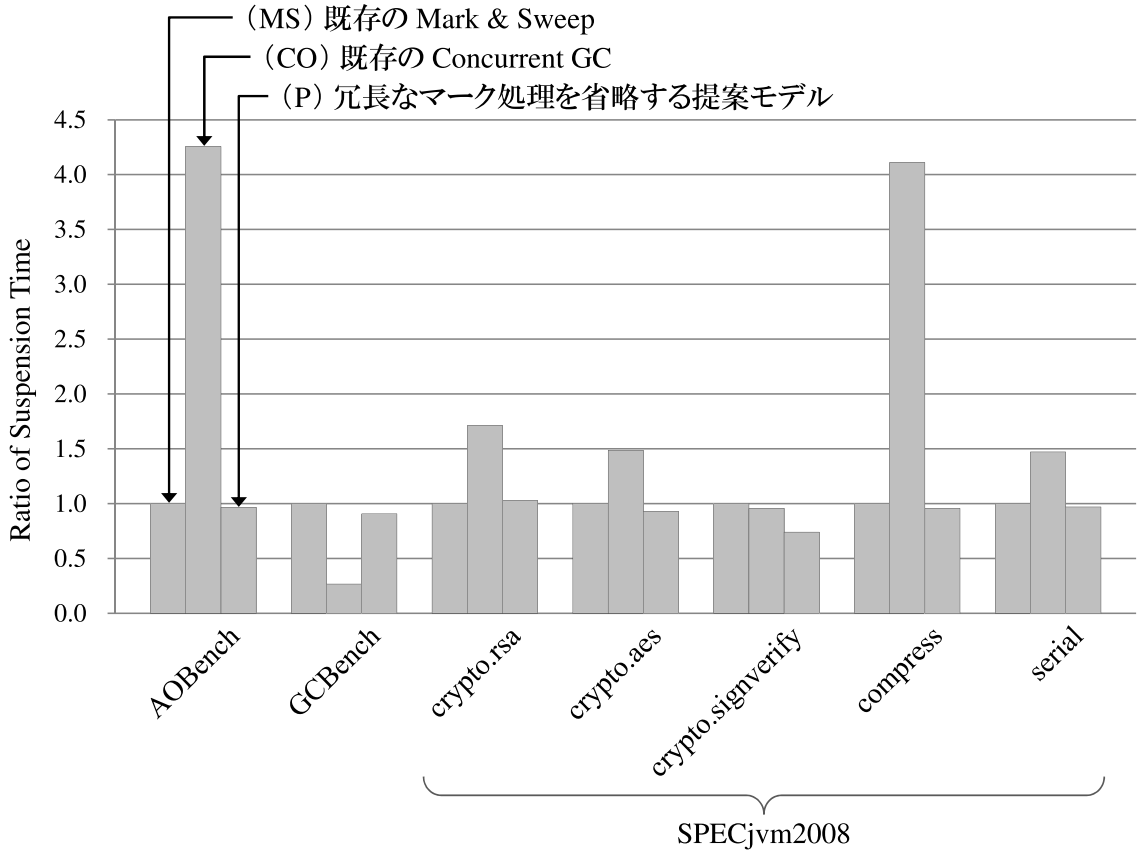

図 28: GC による平均停止時間

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．

なお Concurrent GC を用いた場合，GC の一部の処理はアプリケーションと並行して動作しているため，ユーザ側から見たアプリケーションの停止時間は図 28 に示したものより短くなる可能性がある．しかしシミュレータを用いた評価ではユーザ体感品質について考察することが困難なため，本評価では GC 一回あたりの停止時間の総和を用いて平均停止時間を算出している．そのため，今後はユーザ体感品質に与える影響についても評価可能な環境を構築し，これを用いて本提案手法の有効性を検証していく予定である．

6.3 考察

本節では，以上で示した評価結果から，提案手法による GC の高速化がシステムの全体性能に与える影響について考察する．まず提案モデル (P) と既存モデル (MS) を比較すると，提案モデルは多くのベンチマークプログラムで GC 実行サイクル数を削減

表 4: 専用表検索時のヒット率（%）

	一次検索表		二次検索表	
	(P)	(R)	(P)	(R)
AOBench	52.8	52.8	10.8	11.2
GCBench	50.0	50.0	0.1	0.1
crypto.rsa	48.6	48.7	15.7	16.2
crypto.aes	48.5	48.3	9.9	11.0
crypto.signverify	48.3	48.4	11.1	11.6
compress	53.6	53.3	11.9	13.1
seial	48.3	48.4	9.8	10.5

できており，これに伴って GC による平均停止時間も短縮できている．つまり，GC 実行時におけるシステムのレスポンス低下を緩和できており，応答性が向上していると考えられる．また 6.2.2 項で示したように，提案モデル (P) を用いた場合の総実行サイクル数は既存モデル (MS) とほぼ同程度であり，Concurrent GC で発生するようなサイクル数の増加を防ぐことができている．このことから，提案モデル (P) は既存モデル (MS) と同程度のスループットを保ちつつ，システムの応答性を向上できていることが分かる．

ここで，本提案手法では 5.1.2 節で述べた通り，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．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，二次検索表内の各エントリを一次検索表と同様に LRU に基づいてエントリを追い出すようにした参考モデル (R) を評価し，これを提案モデル (P) と比較することで，提案手法におけるエントリの管理方法の有効性について考察する．

まず，提案モデル (P) と参考モデル (R) における GC 実行サイクル数の評価結果を**図 29** に示す．評価結果を見ると，(P) と (R) の GC 実行サイクル数削減率は同程度となっており，大きな差がないことが分かる．この理由について考察するために，ベンチマークプログラム実行時における各専用表のヒット率をそれぞれ計測した．その結果を**表 4** に示す．この表は，提案モデル (P) と参考モデル (R) における各専用表のヒッ

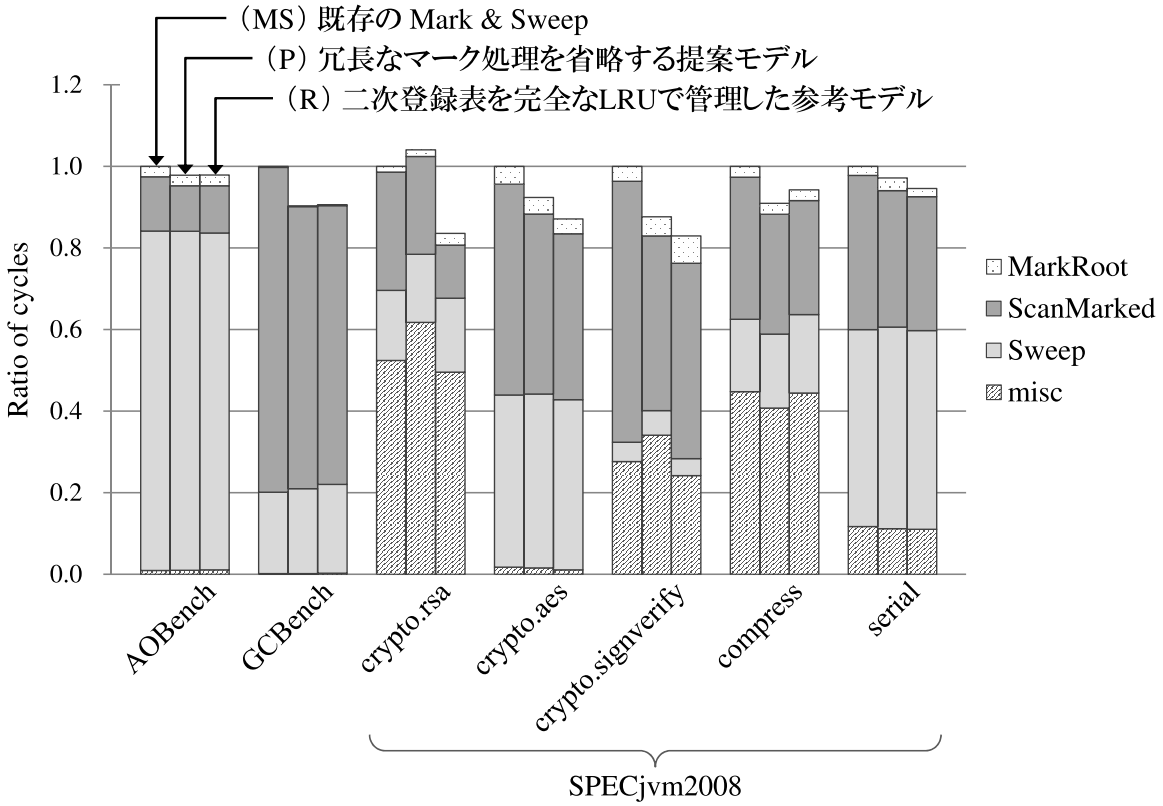

図 29: 完全な LRU 方式を用いた場合との比較

ト率を示している．これを見ると，一次検索表，二次検索表ともに，二つのモデルで同程度のヒット率となっていることが分かる．これは，これら二つのモデルを比較した場合，マーク処理を省略できた回数が同程度であったことを意味している．これにより，これらのモデルにおける GC 実行サイクル数の削減率が同程度になったと考えられる．

また，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%程度となっている．このヒット率について考察するために，マーク済みの全てのオブジェクトを記憶できるように専用表のエントリ数を十分に大きくしたモデルを実装し，この場合のヒット率を調査したところ，一次検索表のヒット率は最大でも約 62%であることが分かった．このモデルではいずれのオブジェクトも表から追い出されることがなく，表に一度登録されたオブジェクトは以降の検索で必ずヒットするため，このモデルにおけるヒット率は理想的な数値を示している．この値と比較しても一次検索表のエントリ数を 50 に制限した本提案手法のヒット率は十分に高いことから，本提案手法では冗長なマーク処理を高い確率で抑制できていると考えられる．

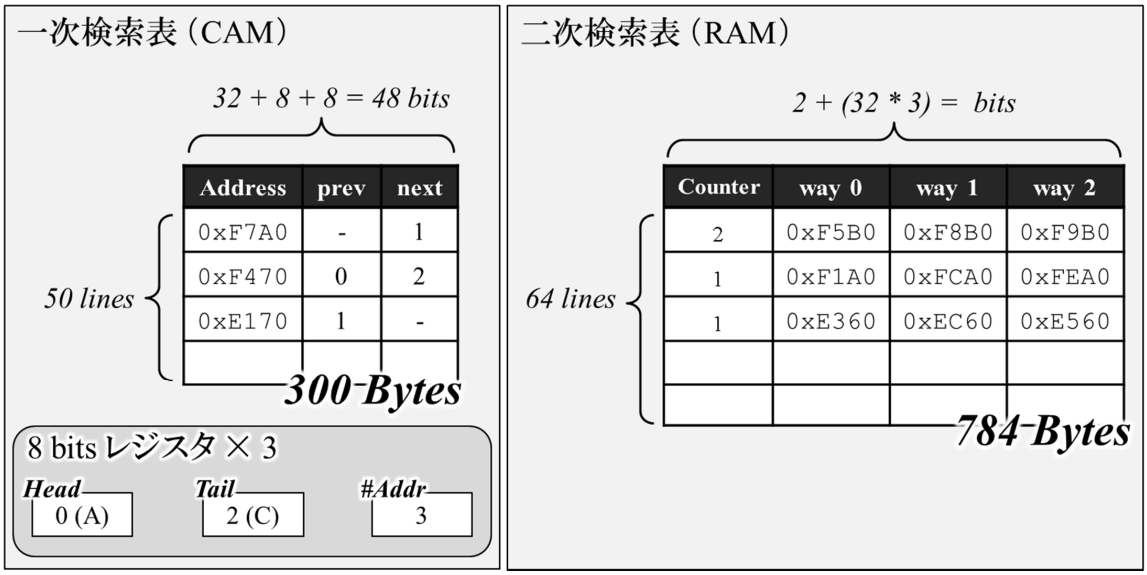

図 30: 専用表のサイズ

結果をまとめると，提案手法を用いた場合，Concurrent GC のようなスループットの悪化を引き起こすことなく，GC 実行サイクル数を削減でき，これに伴ってシステムの応答性も向上すると考えられる．さらに，完全な LRU 方式に基づいて追い出すエントリを決定する参考モデル (R) と比較しても同程度の GC 実行サイクル数を削減できていることから，提案手法は LRU 方式と比較してより単純な制御ロジックで同程度の性能を実現できることが確認できた．

6.4 ハードウェアコストの見積り

本節では，提案モデルの実現のために追加したハードウェアの実装コストについて考察する．ここで，各専用表のハードウェアコストを**図 30** に示す．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ている．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．つまり，一次検索表は幅 48bit，深さ 50 行の CAM で構成できる．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が

必要である．本提案手法では，ウェイ数を 3 としているため，カウンタは 2bit のフィールドで実現できる．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．つまり，二次検索表は幅 98bit，深さ 64 行の RAM で構成できる．

以上のことから，本提案手法は 300Byte の CAM と 784Byte の RAM，および三つの 8 ビットレジスタで構成できることが分かった．このハードウェアコストは合計でも約 1KByte であり，一般的な L1 キャッシュと比較しても十分に小さいことから，本提案手法は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

本論文では，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冗長なマーク処理を省略する．さらに，一次検索表，二次検索表の二つの表を利用し，冗長なマーク処理が頻発しているオブジェクトを優先的に管理することで，マーク処理の省略による効果をより得られるようにした．これにより，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，最大 12.4%の GC 実行サイクル数が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った一方，提案手法ではそのような性能悪化を抑制できており，手法の有効性を確認した．

本研究の今後の課題として，以下の三つが挙げられる．まず一つ目の課題として，ハードウェアの追加に伴って増加すると考えられる消費電力の調査，およびこれを抑制する手法の考察が挙げられる．特に，本提案手法で一次検索表の実装に用いている CAM は消費電力が比較的大きい．そのため，まずこれらの追加ハードウェアに伴って増加する消費電力量を評価し，その結果に基づいて低消費電力で実現可能なハードウェア支援機構を考察する必要がある．

また本論文でも述べたように，現段階の実装では nop 命令を擬似的に専用命令として使用し，これを既存の GC 処理ルーチンを実現している命令列の適切な位置に挿入することで，専用表の操作を実現している．しかし，これらの作業は一般のユーザに

とっては困難である．そのため，二つ目の課題として，専用表を操作するために ISA を拡張し，さらにこれを容易に利用可能とするための GC ライブラリを実装することが挙げられる．これにより，一般のユーザであっても容易に GC 高速化のための機構を利用できるようになると期待できる．

さらに三つ目の課題として，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．本論文で述べた手法では，既存の GC アルゴリズムをベースとして，これに含まれるオブジェクト探索処理のみを支援している．そのため，この処理に要する時間が短いものなど，実行するプログラムによってはハードウェア支援による効果を得られない可能性がある．そこで，Mark & Sweep など既存の GC アルゴリズムに囚われることなく，GC 高速化のための追加ハードウェアをより効率的に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，これまでになかった GC の飛躍的な性能改善の方法を探っていきたい．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の津邑公暁准教授，松尾啓志教授，齋藤彰一准教授，松井俊浩准教授，梶岡慎輔助教，川島龍太助教に深く感謝致します．また，本研究の際に多くの助言，協力をしていただいた松尾研究室，齋藤研究室および松井研究室の方々に感謝致します．特に津邑研究室の方々には本当にお世話になりました．津邑研究室の一員として過ごした 3 年間，時には研究が上手くいかずに落ち込んでしまうことも多々ありました．そんな時でも嫌な顔一つせず，よく話を聞いてくれた橋本高志良氏には本当に感謝しています．独特なセンスから生み出される言葉の数々は，落ち込んでいた私の心をいつも落ち着かせてくれました．また，学会で一緒に発表することが多く，その発表準備などで様々な助言を頂いた柴田裕貴氏にも深く感謝いたします．学部時代の呼び名が信じられないほどに成長し，脅威の問題解決能力で数々の困難を乗り越えられるようになったその姿には，時に感動すら覚えるほどでした．そして，私がこれまでに執筆してきた論文の多くをチェックしていただいた松永拓也氏にも深く感謝いたします．研究分野が全く異なるにも関わらず，頂いたチェックの中には鋭い指摘も多く，論文執筆時の大きな助けとなりました．また普段の生活でも，持ち前の明るさで研究室を盛り上げてくれたことに感謝いたします．さらに，同期だけでなく，津邑研究室で共に活動した後輩にも深く感謝いたします．特に，様々な論文や発表資料のチェックを担当させていただいた竹嶌良氏に心から感謝致します．これまでにチェックさせていただいた合計 2,635 個

の指摘によって，私の文章推敲能力，および文章作成能力は飛躍的に向上したのではないかと思います．最後に，繰り返しになりますが，これまでの3年間ずっと見守っていて下さった津邑公暁准教授に改めて深く感謝致します．この研究室で過ごせたことで,「人に伝える能力」や「問題解決能力」をしっかりと鍛えることができました．時には忙しく大変なこともありましたが，今となっては津邑研究室で過ごせて本当に良かったと思っています．紙面の都合上，その他多くの方に関しては割愛させていただきますが，津邑研究室に所属している方のみならず，これまで一緒に活動してきた全ての方に感謝いたします．同期や，先輩・後輩に恵まれていたからこそ，乗り越えられたことも多かったと思っています．3年間，本当にありがとうございました．

著者発表論文

論文

1. Kei IDEUE, Yuki SATOMI, Tomoaki TSUMURA, Hiroshi MATSUO: “Hardware-Supported Pointer Detection for common Garbage Collections”, Proc. 1st Int’l Symp. on Computing and Networking (CANDAR’13), REGULAR PAPER, Dogo Spa, Japan, pp.134-140 (Dec. 2013)

報文

1. 井手上 慶, 河村 慎二, 津邑 公暁: “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”, 情処研報, Vol.2014-ARC-212, No.1, pp.1-9 (Oct. 2014)
2. 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 (HOKKE-21), Vol.2013-ARC-207, No.27, pp.1-9 (Dec. 2013)
3. 井手上 慶, 里見 優樹, 津邑 公暁, 松尾 啓志: “GC 実行時のポインタ判別コストを削減するハードウェア支援手法の検討”, 信学技報 (SWoPP2013), Vol.113, No.169, pp.19-24 (Jul. 2013)

参考文献

[1] Bornstein, D.: Dalvik Virtual Machine Internals, Google I/O 2008 (2008).

[2] Mozilla Corp.: Firefox OS, http://www.mozilla.org/en-US/firefox/os/.

[3] WebOS-Ports: http://webos-ports.org/wiki/Main_Page.

[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] 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).

[9] Lieberman, H., Hewitt, C. and Hillis, D.: A real-time garbage collector based on the lifetimes of objects, *Communications of the ACM*, Vol. 26, pp. 419–429 (1983).

[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] Printezis, T. and Detlefs, D.: A Generational Mostly-concurrent Garbage Collector, *Proc. The 2000 Int'l Symp. on Memory Management (ISMM 2000)*, ACM Press, pp. 143–154 (2000).

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

[13] New Unified Environment Research Project: http://www.nue.org/nue/.

[14] 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).

[15] Wilson, P. R., Johnstone, M. S., Neely, M. and Boles, D.: Dynamic Storage Allocation: A Survey and Critical Review, Springer-Verlag, pp. 1–116 (1995).

[16] Bobrow, D. G., Burchfiel, J. D., Murphy, D. L. and Tomlinson, R. S.: TENEX, a Paged Time Sharing System for the PDP - 10, *Commun. ACM*, Vol. 15, No. 3,

pp. 135–143 (1972).

[17] 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).

[18] Boehm, H.: An Artificial Garbage Collection Benchmark, http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench.html.

[19] Fujita, S.: Ambient Occlusion Benchmark, http://code.google.com/p/aobench/.

[20] SPEC.: SPECjvm2008 benchmarks, http://www.spec.org/jvm2008/ (2008).

[21] Brash, D.: Recent additions to the ARMv7-A architecture., *ICCD*, IEEE (2010).

[22] ARM Ltd.: RealView Platform Baseboard Explore for Cortex-A9 User Guide, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0440b/index.html.