卒業研究論文

組込みシステムの性能向上を目的とした ハードウェア支援によるGC高速化手法の検討

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学 工学部 情報工学科
平成 20 年度入学　20115074 番

里見 優樹

平成 24 年 2 月 8 日

組込みシステムの性能向上を目的としたハードウェア支援による GC 高速化手法の検討

里見 優樹

内容梗概

現在，スマートフォンを初めとするモバイル機器の普及により組込みシステムにはさらなる高性能化が求められている．高い性能を実現するためには，高パフォーマンスのプロセッサをもってこれに応えることが重要となるが，クロック周波数の上昇に伴う消費電力の増大により，モバイル機器にとって重要なもう一つの要素であるバッテリの持続時間が犠牲になってしまう．モバイル機器において低消費エネルギーを実現するための手法としては，主に待機時間の消費電力削減に着目したものが多いが，稼動時の消費電力を削減できれば，さらなる低消費電力化に繋がると考えられる．

さて，汎用計算機と比較して小容量のメモリしか搭載できないこれらの機器では，メモリ管理を適切に行う必要があり，特にガーベジコレクション (Garbage Collection: GC) の働きが重要となる．しかし，GC は機器の稼働時における性能悪化の大きな要因となることも知られており，GC の高速化は組込みシステムの性能向上を目指す上での重要課題である．しかし，GC に関する研究はそのほとんどがソフトウェア面からの高速化についてのものであり，ハードウェア面からの高速化はこれまであまり議論されてこなかった．

そこで本研究では，GC の高速実行をハードウェア的にアシスト可能なプロセッサ構成方式を提案する．GC にかかるコストを削減することで，クロック周波数を大きく上昇させることなく，アプリケーションに対する高い即応性と低消費電力の両立を実現するプロセッサを実現することを目標とする．

本論文では，まず既存の GC アルゴリズムを評価し，動作のボトルネックを調査した．調査の結果，コールスタックの走査に多くのサイクル数を要していることが分かった．そこで，コールスタック内に格納されているヒープ領域中のオブジェクトへのポインタを表に記憶しておくことで，コールスタックの走査を高速化する手法を提案する．この手法により，コールスタック走査を省略し，それまでこの処理にかかっていたサイクル数の削減を図る．

また，提案手法がもたらす効果について考察した．表の操作にかかるコストを見積もり，提案手法によって削減されると思われるサイクル数と比較した．その結果，提案手法を適用することで多くのサイクル数の削減が見込めることが分かった．

組込みシステムの性能向上を目的としたハードウェア支援による GC 高速化手法の検討

目次

1 はじめに

現在，スマートフォンを初めとするモバイル機器の普及により組込み機器にはさらなる高性能化が求められている．高い性能を実現するためには，高パフォーマンスのプロセッサをもってこれに応えることが重要となる．現状，各メーカーでは，1GHz を超える高クロックプロセッサを利用することでこれに対応しようとしている．しかし，クロック周波数の上昇に伴う消費電力の増大によりモバイル機器にとって重要なもう一つの要素であるバッテリの持続時間が犠牲にされている．モバイル機器において低消費電力を実現するための手法としては，主に待機時の消費電力削減に着目したものが多く研究されているが，稼動時の消費電力を削減できればさらなる低消費電力化に繋がると考えられる．

さて，モバイル機器で最もよく利用されるアプリケーションの一つにウェブブラウザがあるが，このウェブブラウザのパフォーマンスを低下させている処理の主たるものは，**ガーベジコレクション (Garbage Collection: GC)** である．汎用計算機等と比較すると小容量なメモリしか搭載できないモバイル機器では，全体の処理に占める GC の割合が大きい．大きい．特にブラウザ上の JavaScript エンジンの実行時間において GC に要する時間が占める割合は非常に大きいことが知られており，十数%から数十% にも及ぶとも言われている．

以上のことから，GC の高速化は組込みシステムの性能向上を目指す上での重要課題であると言えるが，GC に関する研究はそのほとんどがソフトウェア面からの高速化についてのものであり，ハードウェア面からの高速化はこれまであまり議論されてこなかった．

そこで本研究では，GC の高速実行をハードウェア的にアシスト可能なプロセッサ構成方式を提案し，クロック周波数を大きく向上させることなく，消費電力を低く抑えつつアプリケーションに対する高い即応性を実現するプロセッサを実現することを目標とする．本論文では実際に GC の振る舞いを調査し，その結果を基にハードウェア支援による GC 高速化手法を考察する．

以下，2 章では研究背景として，組込みシステム，GC，GC のハードウェア支援の既存研究について説明する．3 章では，GC の動作のボトルネックを調査しハードウェア支援手法を検討するための予備評価を行う．4 章では予備評価の結果から GC の高速実行が可能となるハードウェア支援手法を提案し，5 章で提案手法の効果についての考察を行う．そして 6 章で結論を述べる．

2　研究背景

本章では，本研究の背景となる，組込みシステム，GC，GC のハードウェア支援の既存手法について説明する．

2.1　組込みシステム

現在，身の回りの多くの機器がコンピュータシステムによって制御されている．例を挙げると，テレビ，エアコン，デジタルカメラ，携帯電話といった家電製品から，自動車，船舶，航空機といった輸送機器まで様々である．これらのような機器に組み込まれ，特定の機能を実現するコンピュータシステムを**組込みシステム**と呼ぶ．

組込みシステムには機器のサイズの制約や，独自ハードウェアの開発にかかるコスト等の理由から，汎用計算機と比較して限られたリソースしか搭載できない．しかしながら，技術の進歩によるハードウェアの性能単価の下落とともに組込みシステムの適用される範囲は拡大してきた．それに伴い，組込みシステムに対してより複雑で多岐に渡る機能の実現が求められるようになった．中でもそうした傾向が顕著であるのがモバイル機器である．スマートフォンの普及もあり，モバイル機器に汎用計算機並みのリッチコンテンツを処理することが求められている．さらに，モバイル機器には単に高い処理性能だけでなく，省電力化によるバッテリ持続時間の向上も求められている．

こうした性能要求に応えるためにはプロセッサの高性能化の他に，GC の高速化もまた重要である．なぜなら，汎用計算機と比較して小容量のメモリしか搭載できないモバイル機器では，メモリ管理の性能が機器の性能に大きく影響を与えてしまうからである．例えば，モバイル機器でよく使用されるアプリケーションの一つにウェブブラウザがあるが，このウェブブラウザ上で動作する JavaScript エンジンの実行時間に占める GC の割合は十数%から数十%に及ぶとも言われている．そのため，GC を高速化できればアプリケーションの動作に必要なクロック周波数を相対的に低く抑えることが可能になり，さらなる低消費電力化にも繋がる．

2.2　ガーベジコレクション

“GC” とは，“Garbage Collection” の略であり，日本語では “ゴミ集め” という意味である．GC において “ゴミ” というのは，プログラム実行中に動的に確保されたメモリ領域のうち，不要になった領域のことである．そして，その領域を自動的に解放し，

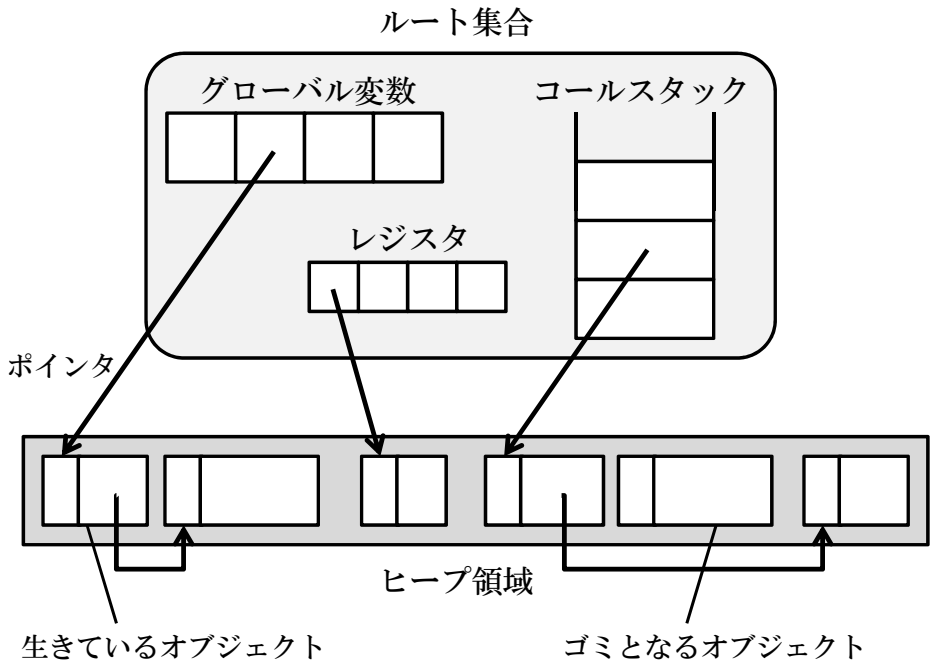

図 1: プログラム実行時のメモリの様子

空き領域としてプログラムが再び使用可能にするのが GC の役割である．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り，プログラムのより本質となる部分の設計に注力できる．現在，Java，JavaScript，Lisp，Perl 等，C や C++を除くほとんどの言語処理系において GC は実装されている．

ここで，本論文で想定する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．メモリ領域の中でも，GC が対象とするのは，ヒープ領域内のオブジェクトである．まず GC におけるオブジェクトというの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り，GC はこのオブジェクトに対して，移動や破棄といった操作を行う．またヒープ領域とは，プログラムが動的に確保することのできるメモリ領域である．メモリ確保のためのルーチンであるアロケータはこのヒープ領域からオブジェクト単位でメモリ領域を確保する．そして，ヒープ領域内に確保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やレジスタ等，プログラムから直接参照できる領域に格納される．これらの，プログラムから直接参照可能な領域の集合をルート集合と呼ぶ．ヒープ領域中に確保されたオブジェクトは，基本的にルート集合からポインタを辿ることで参照可能である．また，確保されたオブジェクトがヒープ内の

他のオブジェクトへのポインタを保持することもあるため，ルート集合から他のオブジェクトを経由して参照可能なオブジェクトも存在する．こういったルート集合から直接または他のオブジェクトを経由して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．

しかし，ポインタが書き換えられるとルート集合からポインタを辿っても参照できないオブジェクトが発生してしまうことがある．ルート集合からポインタを辿れないということは，プログラムからもう二度と参照されないということである．GC はこの様なオブジェクトをゴミと判断する．ゴミとして扱われたオブジェクトは GC 実行時に破棄され，割り当てられていたメモリ領域が解放され，空き領域として再び使用可能となる．オブジェクトの生成やポインタの書換えを行うことで，オブジェクト間の参照関係を変化させるものを**ミューテータ**という．

このゴミとなる領域を検出し，解放するための GC アルゴリズムは多数存在する．その中でも全ての基本となる 3 つのアルゴリズムとして，Mark & Sweep [1] ，Copying [2] ，Reference Counting [3] が存在する．これまでに開発されてきた全ての GC アルゴリズムは，これら 3 つのアルゴリズムを改良したもの，あるいはそれぞれを組み合わせたものである [4] ．以下，これら 3 つのアルゴリズムの動作とその特徴について述べる．

2.2.1 Mark & Sweep

Mark & Sweep は日本語では目印付け法と呼ばれるもので，生きているオブジェクトにマークを付けてゴミかどうかを判別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例を**図 2** に示す．図中の色付きのオブジェクトはマーク済のオブジェクトを表している．まず，ルート集合を走査し，そこから直接参照できるオブジェクトをマークする [図 2(a)]．そして，マークされたオブジェクトから参照できるオブジェクトをマークしていき，ポインタを持たないオブジェクトに辿り着くまでこれを繰り返す [図 2(b)]．これを**マークフェーズ**と呼ぶ．マークフェーズが終了した時点でマークのついていないオブジェクトはルートから辿ることのできないオブジェクト，つまりゴミであると判断できる．そこで，GC は次にヒープ領域全体を走査しマークのついていないオブジェクトを解放する．これを**スイープフェーズ**と呼ぶ．Mark & Sweep はこれら 2 つのフェーズで構成される．

しかし，このアルゴリズムでは，ゴミとなるオブジェクトをただ解放するだけで，オブジェクトの移動等は行わないため，空き領域のオブジェクトのサイズが統一されていない場合，ヒープ領域内でフラグメンテーションが生じてしまう [図 2(c)]．フラグ

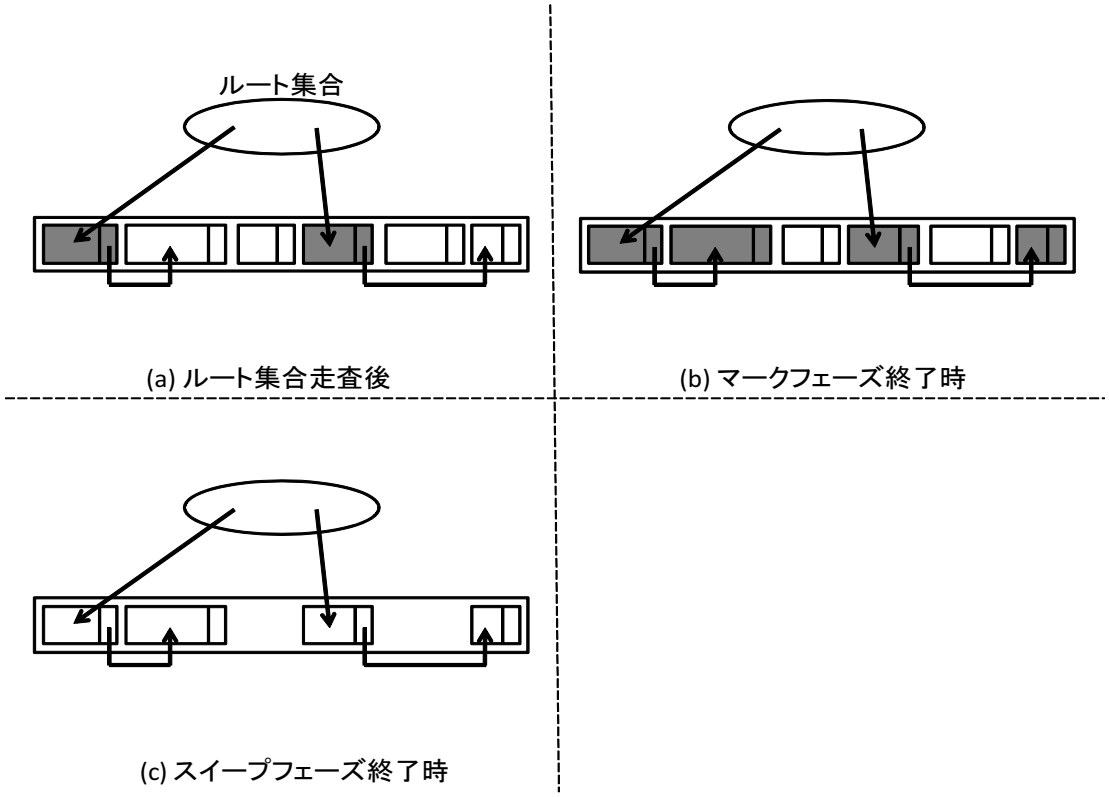

図 2: Mark & Sweep

メンテーションが起こると，ヒープ領域内の空き領域の合計は十分であっても，その 1 つ 1 つが小さいためにメモリ領域の割り当てができないといった問題が生じてしまう．

2.2.2 Copying

Copying は，生きているオブジェクトだけを別の場所に移すというアルゴリズムである．このアルゴリズムの動作例を**図 3** に示す．このアルゴリズムでは，ヒープ領域をオブジェクトの移動元と移動先の 2 つの領域に分割して使用する．[図 3(a)]．この分割された領域を，それぞれ from-space，to-space と呼ぶ．from-space がオブジェクトの移動元，to-space が移動先である．新しくオブジェクトが生成された場合，from-space のメモリ領域を割り当てる．そして，from-space に十分な空き領域が無くなり，メモリ領域の割り当てに失敗すると GC が実行される．

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 [図 3(b)]．コピーが終わった時点では，to-space にコピーされたオブジェクトが持つポインタは from-space のオブジェクトを指したままの状態なので，to-space のオブジェクトを指すようにポインタを修正する．また，ルート集合の持つポインタも同様に修正する [図 3(c)]．以上により，ポインタの修正を含めた to-space への移動が

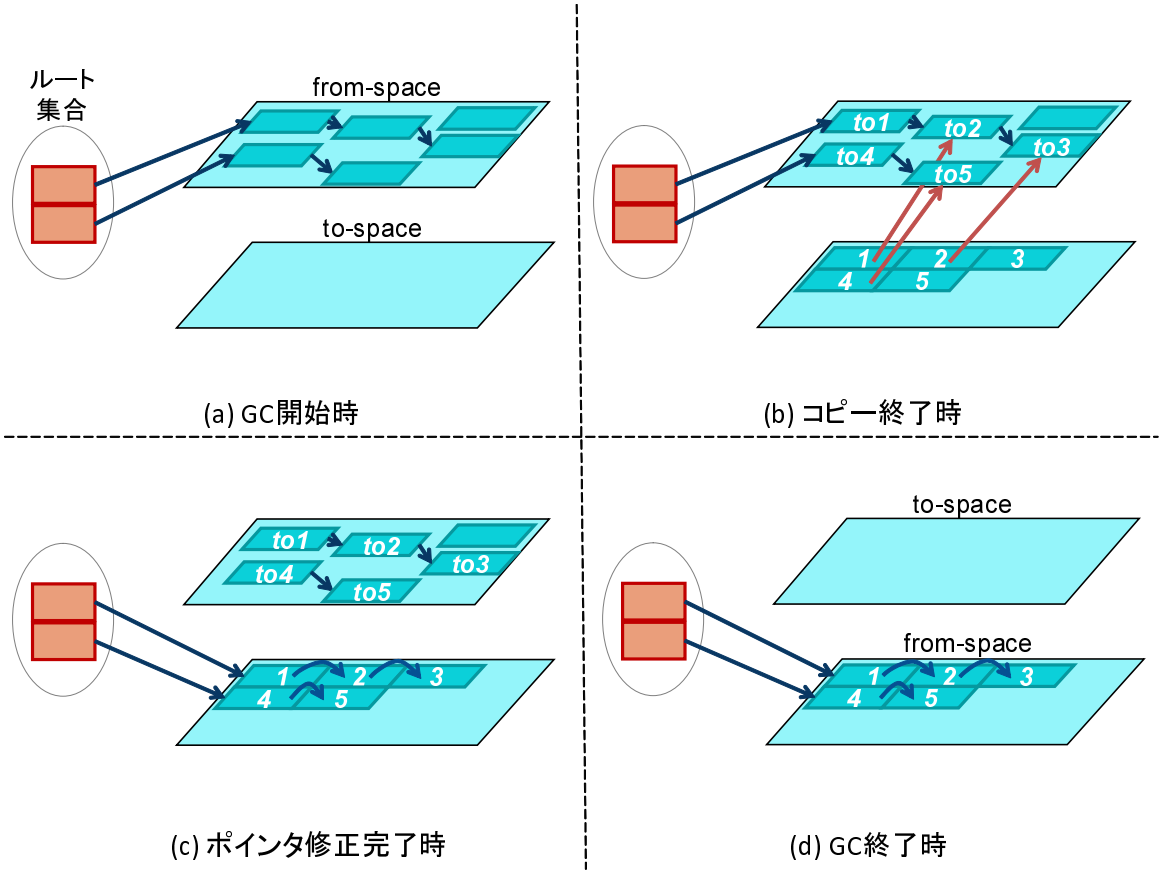

図 3: Copying

完了したので， from-space 全体を解放する．そして，以降は to-space を from-space として扱い，from-space を to-space として扱う [図 3(d)].

このアルゴリズムでは，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，フラグメンテーションは発生しないという利点がある．また， Mark & Sweep の解放処理ではヒープ全域を走査する必要があったが， Copying では走査をせずに from-space ごと解放するだけなので，解放処理が速い．しかし，ヒープ領域を半分しか使用できないので，ヒープ領域の使用効率が悪いという欠点がある．

2.2.3 Reference Counting

Reference Counting は，どこからも参照されていないオブジェクトはゴミである，という考え方に基づいたアルゴリズムである．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．そして，参照カウンタの値が 0 になった，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メモリ領域を解放する．Reference Counting では，各オブジェクト自体がカウンタを持つので，オブジェクトは自らがゴミであるかどうかを判断できる．そのため，他のアルゴリズムの様にミューテータが明示的に GC を呼び出すことはなく，オブジェクトがゴミとなると同時に解放される．つまり，メモリ領域がゴミで使用されることがない．

しかし，カウンタは最大で，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．そのため参照カウンタには多くのビットが必要となり，全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現実的ではない．一方で，ほとんどのオブジェクトの被参照数は 1 であることが経験的に知られている．そこで，実際には全てのオブジェクトに参照カウンタを用意することはせず，被参照数が 0 か 1 かを表すカウンタを設ける．そして，被参照数が 2 以上となったオブジェクトは参照表と呼ばれる表へ登録する．この参照表には，オブジェクトのアドレスと被参照数が登録される．これにより，参照カウンタの設置によるメモリ領域の使用効率悪化を緩和することができる．

この方式では，ミューテータがポインタを書換えた時に，参照表への操作や，インクリメント，デクリメントを行うため，GC によるミューテータの実行停止が分散され，それにより，最大停止時間を抑えることができる．しかし、通常ポインタの書換えは頻繁に発生するので，その度に表への操作を行うことは，スループットを悪化させてしまう．

2.3 既存のハードウェア支援手法

本節では，ハードウェア支援による GC 高速化の二つの既存研究について説明する．

2.3.1 SILENT

SILENT [5] とは，NUE プロジェクト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 を採用しており，GC とアプリケーションはそれぞれ別のプロセスとして動作している．GC プロセスは 8 つ動作しており，2 つはマーク用，残り 6 つはスイープ用のプロセスとなっている．そして，これらの GC プロセスとミューテータであるアプリケーションプロセスが並行に動作する．

ミューテータと GC プロセスを並行に動作させると，生きているオブジェクトのマーク漏れという問題が発生してしまう．**図 4** にこの問題が発生する場合の動作例を示す．

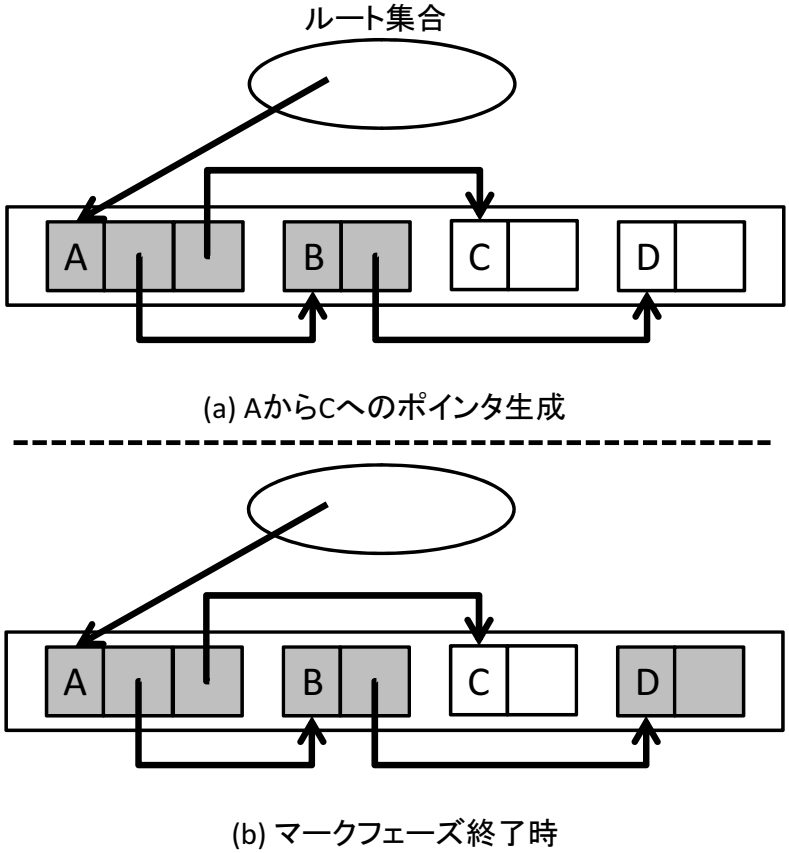

図 4: ポインタの書き換えによるマーク漏れ

GC プロセスがマークフェーズを実行中，オブジェクト B までマークを完了した時点で，ミューテータ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 [図 4(a)]．しかし，GC プロセス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 [図 4(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，本来解放してはいけないオブジェクトであるにも関わらず解放されてしまう．

このようなマーク漏れによる誤った解放を防ぐために，SILENT ではライトバリアという手法が用いられる．ライトバリアとは，書き込みを検知してその書き込みによるデータ不整合を発生させない様に同期処理を行うことである．SILENT では，ミューテータによるポインタの書き換えを検知し，それを GC プロセスに知らせることで，ライトバリアを行っている．

具体的には，マーク済オブジェクトが未マークオブジェクトを参照する様にポインタを書き換えたとき，ミューテータ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全体として多くの時間がかかる処理となってしまう．そこで SILENT では，このライトバリアをハードウェアサポートによって高速化している．SILENT では，ライトバリアが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されている．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で，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なため，実行速度の面で非常に最適化された動作を記述できる．そのため，マイクロプログラムで記述されたライトバリアは非常に高速に動作する．

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 マイクロ秒以下に抑えられている．ただし，この手法は Lisp マシンという特殊な計算機上でのみ実現されているため，多様なプラットフォームが存在する組込みシステムに移植するのは困難である．

2.3.2 Network Attached Processing

Network Attached Processing(NAP) [6] 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．NAP では，SILENT と同様に，GC スレッドとミューテータである Java スレッドが並行に動作している．GC スレッドとミューテータを並行動作させるには何らかの同期処理が必要であるが，NAP では同期処理にリードバリアを用いている．リードバリアとは，読み出しを検知してその読み出しによるデータ不整合を発生させない様にに同期処理を行うことである．NAP では，2 種類のリードバリアをハードウェア支援によって実現している．どちらのリードバリアも，ミューテータによるポインタの読み出しを検知するものであるが，その後の動作が異なる．

Pauseless GC では，Mark & Sweep と同様に GC スレッドがマークを行うが，この時，巡回したポインタにもマークを行う．Java ではオブジェクトが 8 バイト境界に配置される，つまりオブジェクトを指すポインタの値が 8 の倍数であるため，ポインタをビット列で表すと下位 3 ビットが常に 0 となることを利用し，そのうちの 1 ビットをマーク用のタグとして利用する．これを Not Marked Through(NMT)-bit と呼び，それが 1 であるならば，GC スレッドがそのポインタを巡回済みであることを示す．そし

て，GC スレッドがマークフェーズを実行中，ミューテータ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，そのポインタの NMT-bit をチェックし，もし未巡回であれば NMT-bit を立て，そのポインタを GC スレッドに通知する．この様に，マークフェーズではミューテータが NMT-bit のチェックをすることでリードバリアが行われる．これが 1 つ目のリードバリアである．こうして，ミューテータがマーキングに協力することで，GC スレッドはミューテータのポインタ書き換えによる影響を考慮することなく動作することができる．

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の追加により実装されている．そして，この命令はミューテータがポインタの参照オブジェクトをロードする際に実行される．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 サイクル程度と小さいため，非常に高速にリードバリアが実行される．

2 つ目のリードバリアは，マーク完了後の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内で実行される．NAP ではヒープ領域をページ単位で区切っているため，ゴミとなるオブジェクトのみのページがあれば，まずはそのページを解放する．だが，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数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る．しかし，このオブジェクトの移動をミューテータと並行して実行することは非常に難しい．なぜならば，移動させる際に，移動対象のオブジェクトを指すポインタを，移動先を指す様に全て書き換える必要があるからである．このポインタの書き換えをリマップと呼ぶ．そこで，リロケーション後すぐにはリマップを行わず，移動元と移動先アドレスの対応表を作っておく．そして，移動元のページには GC スレッドが GC-protect という特殊な保護をかける．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の修正が行われる．この様に，リロケーションフェーズではページに特殊な保護かけることでリードバリアを行う．これが 2 つ目のリードバリアである．

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る．この GC-trap 内ではロードしたポインタのリマップ処理が行われる．

GC 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

表 1: 評価環境

マシン	TS-7200
プロセッサ	ARM920T
周波数	50 MHz
メモリ	32 MB
OS	Linux 2.4.26-ts9

ずにマークフェーズに入る．そして，GC 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，リマップのやり残しは発生しない．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．

この様に，NAP ではリードバリアを行うことでミューテータ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，NAP はその名が示す通りネットワーク接続型のリモート計算機として実現されることが想定されており，そのためホストマシン間との通信が性能のボトルネックとなってしまう．また，モバイル端末の様な不安定な通信下での使用が想定される機器において NAP を使用することは現実的ではない．

3 GC アルゴリズム内ボトルネックの調査

2.3 節で述べた既存手法は特定言語の実行に特化していたり，同期処理に絞ってハードウェア支援していたりと，その着眼点，用途は限定的である．そこで GC アルゴリズムの構成処理要素，例えば Mark & Sweep のルート集合の走査，ヒープ領域のスイープ等，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．支援手法を考察するにあたり，まず GC の動作について予備評価を行った．評価は GC アルゴリズムの構成処理要素毎の実行時間の内訳を調査することで行う．またその結果からアルゴリズムのボトルネックについて考察する．

3.1 評価環境

評価にはフルシステムシミュレータである Simics[7] を用いた．想定するシステムの環境を**表 1** に示す．プロセッサには組込み機器向けに広く用いられる ARM アーキテクチャを選択した．ARM920T は，32 ビットの RISC プロセッサ，TS-7200 は ARM920T を搭載するシステム開発用シングルボードコンピュータである．

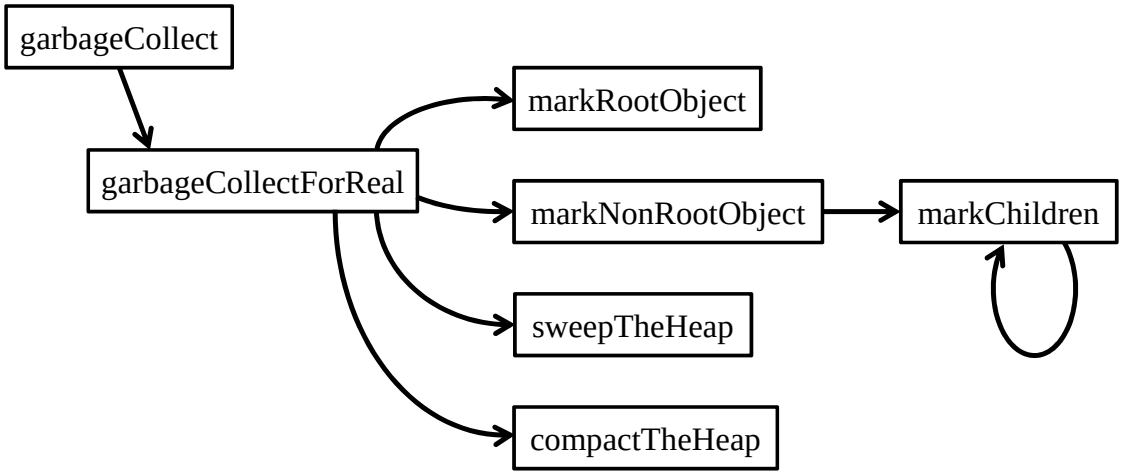

図 5: garbageCollect 関数から呼び出される主な関数

本評価では，JavaVM の一種である KVM(Kilobyte Virtul Machine) を使用する．KVM は実行時に必要なメモリ量が約 160KB と非常にコンパクトであり，携帯電話等に用いられる組込み用 JavaVM である．また KVM では，GC アルゴリズムに Mark & Sweep が採用されている．他の手法でも広く採用されている Mark & Sweep アルゴリズムについて GC の実行を高速化するハードウェア機構を考察することで，多くの GC アルゴリズムの高速化に繋がると考えられる．

次に，KVM の GC アルゴリズムの具体的な実装について述べる．まず，ミューテータが新しいオブジェクトを作成する時に，free list と呼ばれる空き領域のリストを調べ，オブジェクトの確保に十分な空き領域のかたまり (チャンク) が存在するか否かを判断する．そして空き領域が存在しなかった場合，garbageCollect 関数が呼び出され，GC の処理が開始される．garbageCollect 関数から呼び出される主な関数のコールグラフを**図 5** に示す．garbageCollect 関数は，GC が既に実行中であるかを確認し，実行中でなかった場合，現在実行中のスレッドを停止するための処理を行った後，garbageCollectForReal 関数を呼び出す．この関数は GC の処理を実行する関数で，この関数からマークやスイープ等の各処理に相当する関数が呼び出される．garbageCollectForReal 関数の中でまず初めに呼び出されるのが markRootObject 関数である．この関数はルート集合を走査し，その中からヒープ領域中のオブジェクトへのポインタを探索する．そして，発見したポインタが指すオブジェクトをマークする．具体的には，各オブジェクトが持つヘッダ内のそのオブジェクトがマーク済みかどうかを表すビット (MARK BIT) を立てる．markRootObject 関数の実行が終了すると，次に markNonRootObject 関数が呼び

出される．この関数はヒープ領域内の markRootObject 関数でマークされたオブジェクトを探索する．そして，発見したマーク済オブジェクトから参照可能なオブジェクトをマークするために markChildren 関数を呼び出す．この関数は自分自身を再帰的に呼び出すことにより，markNonRootObject 関数によって発見されたマーク済オブジェクトから参照可能なオブジェクトを順次マークしていく．以上でヒープ領域中の生きているオブジェクトに対するマークが完了する．

次に sweepTheHeap 関数が呼び出され，マークフェーズからスイープフェーズへと処理が移行する．sweepTheHeap 関数はヒープ領域全域を走査し，オブジェクトを探索する．そして検出したオブジェクトの MARK BIT をチェックし，MARK BIT が 0 ならばそのオブジェクトを空き領域として解放し free list に追加する．一方，MARK BIT が 1 だった場合は MARK BIT を 0 にリセットする．ここまでが Mark & Sweep に相当する処理である．そして，GC を実行してもミューテータに要求されている大きさのメモリ領域を確保できなかった場合は，compactTheHeap 関数を呼び出しヒープ領域のコンパクションを行う．コンパクションとはオブジェクトの移動を行い，細分化された空き領域をまとめる処理である．

KVM 上で実行するプログラムには，Minibenchmark を使用した．これは各スレッドに文字列操作を仕事として与えるマルチスレッドプログラムである．

3.2 評価結果

GC アルゴリズムの構成要素に相当する関数の実行サイクル数を計測し，その処理時間の内訳を評価した．サイクル数は，各関数の開始アドレスと終了アドレスにプログラムカウンタが到達した時点の CPU サイクル数をシミュレータから取得し算出した．評価結果のグラフを**図 6** に示す．グラフは各関数内の処理時間の内訳を表しており，それぞれの関数の総実行サイクル数を 1 として正規化した．まず，garbageCollect 関数を見ると，実行サイクル数のほとんどを garbageCollectForReal 関数の実行サイクル数が占めていることが分かる．

そしてその garbageCollectForReal 関数の実行サイクル数は主に markRootObject 関数，markNonRootObject 関数，SweepTheHeap 関数の三つの関数の実行サイクル数が占めている．特に markRootObject 関数の実行サイクル数が GC 全体，つまり garbageCollect 関数の実行サイクル数に対して占める割合は約 65%であり，多くのサイクル数を要していることが分かる．

次に markRootObject 関数を見ると，この関数からは markThreadStack 関数と check-

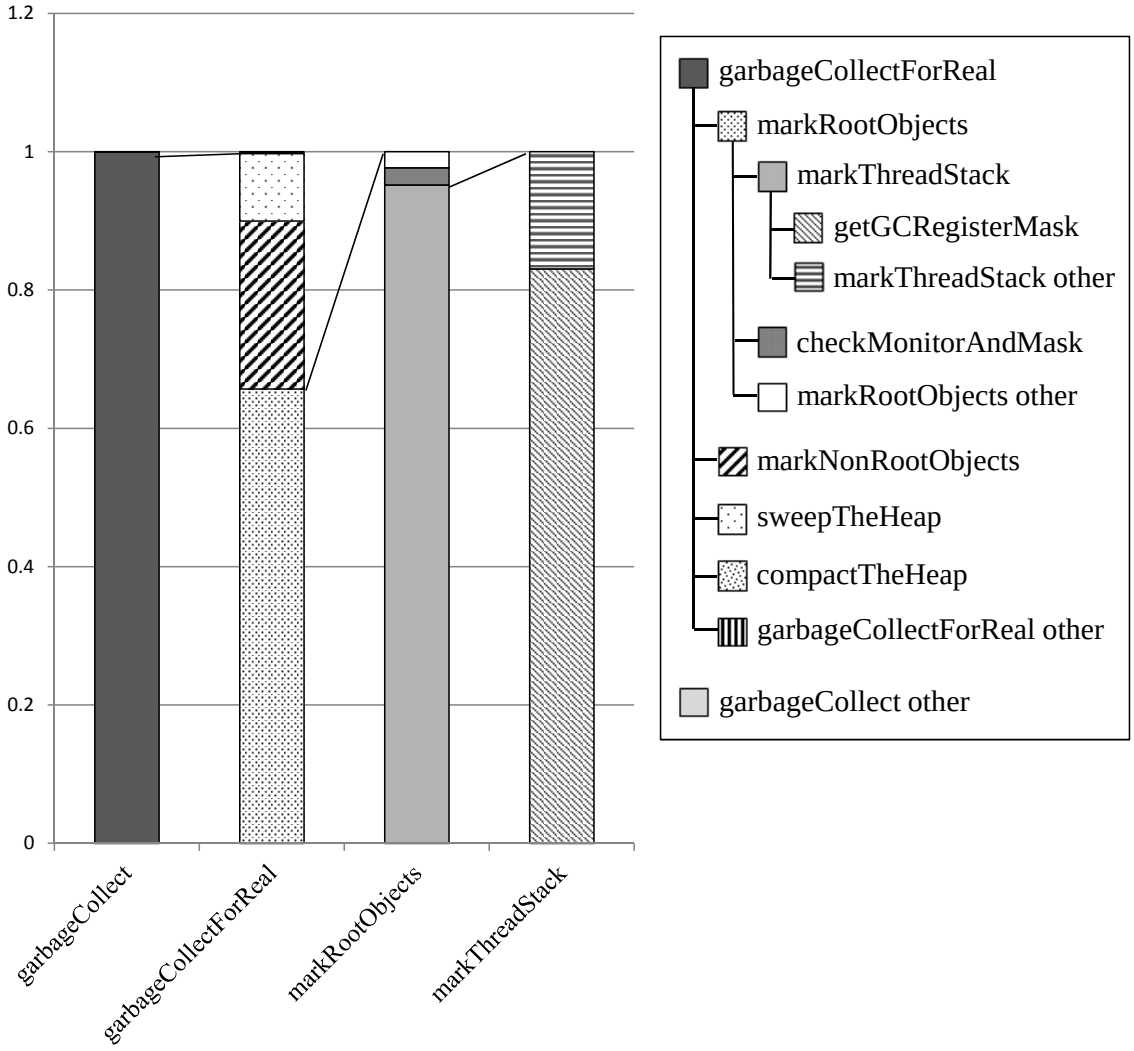

図 6: GC アルゴリズムを構成する関数の実行時間の内訳

MonitorAndMark 関数が呼び出されている．markThreadStack 関数は各スレッドが持つコールスタックを走査して，コールスタック内に格納されているヒープ領域中のオブジェクトへのポインタを発見し，そのポインタが指すオブジェクトをマークする関数である．checkMonitorAndMark 関数はスレッドの同期機構であるモニタを監視する関数である．これらの内，markThreadStack 関数の実行サイクル数が markRootObject 関数の実行サイクル数の約 95%を占めており，ルート集合の中でも特に，コールスタックの走査に多くのサイクル数を要していることが分かった．

最後に，markThreadStack 関数を見ると，この関数からは getGCRegisterMask 関数が呼び出されている．この関数は GC の対象となるオブジェクトへのポインタがス

タックフレーム内のどこに存在するかを表すビット列 (スタックマップ) を作成する．markThreadStack 関数ではこのスタックマップにしたがってフレーム内でポインタが格納されている場所を特定し，そのポインタを辿ってオブジェクトにマークする．グラフから getGCRegisterMask 関数の実行サイクル数が markThreadStack 関数の実行サイクル数の約 83%を占めていることが分かる．これは garbageCollect 関数全体の実行サイクル数の約 52%に相当する．

以上の計測結果から Mark & Sweep アルゴリズムではルート集合，特にコールスタックの走査に多くのサイクル数を要していることが分かった．getGCRegisterMask 関数の実行サイクル数が GC 全体の約 52%を占めていることから，コールスタックの走査に多くの時間がかかるのは GC 実行時にスタック内に含まれるポインタを探索する必要があるためだと考えられる．

4　ハードウェア支援による GC 高速化手法

前章の GC アルゴリズムの調査の結果判明した，GC の動作におけるボトルネックを高速化可能なハードウェア機構を提案する．本章では，提案手法，及びその動作モデル，また実装において考慮すべき点について述べる．

4.1　コールスタック走査の高速化

前節で示した GC アルゴリズムの調査の結果，コールスタックの走査に非常に多くのサイクル数を要していることが分かった．これはコールスタックを走査する際，スタックに格納されている値が GC の対象となるヒープ領域へのポインタであるか否かを判別する必要があるためだと考えられる．

そこで，コールスタック内に格納されているヒープ領域へのポインタを表に記憶しておくことでコールスタックの走査を高速化する手法を提案する．具体的には，コールスタック内のヒープ領域へのポインタを，高速にシーケンシャルな検索が可能なバッファメモリに登録することでポインタの検索を高速化する．これにより GC 実行時のコールスタックの走査を高速化できる．

4.2　動作モデル

本節では，提案手法を実現するために必要となる機構，及びその動作について述べる．提案手法を実現するために，ポインタを一時的に登録しておく表 (一時登録表) と，コールスタックに格納されているポインタを常に保持する表 (参照表) の二つの表を追

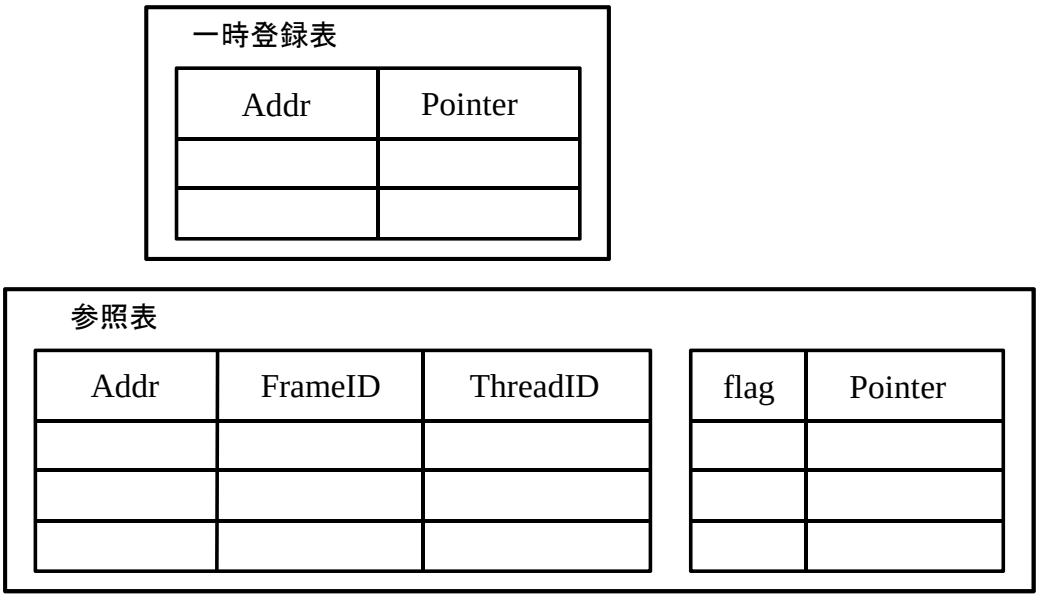

図 7: 一時登録表と参照表の構成

加する．表の構成を**図 7** に示す．このうち一時登録表は，ポインタが格納されているスタック上のアドレスを保持する Addr，スタックに格納されているヒープ領域へのポインタを保持する Pointer の二つのフィールドを持つ．参照表は大きく分けて，エントリの識別情報を格納する部分と GC 実行時に必要とされる情報を格納する部分の二つから成る．このうち前者は，Addr，ポインタがどのスタックフレームに格納されているかを示す FrameID，ポインタがどのスレッドのコールスタックに格納されているかを示す ThreadID の各フィールドから成る．FrameID にはエントリ登録時にフレームポインタ (fp) が指しているアドレスを利用する．また，ThreadID にはスレッド ID を利用する．ここで，スレッド ID は OS や VM から各スレッドに割り当てられるユニークな数値であると仮定する．また，これらの値からポインタの値の変更，削除を行う対象のエントリを特定する必要があるため，参照表のこの部分は連想検索が可能な CAM (Content Addresable Memory) で実装されると仮定する．後者の GC 実行時に必要とされる情報を格納する部分は，そのエントリにポインタが登録されているかどうかを表す flag，Pointer の各フィールドから成る．flag の値が 1 の時はポインタが登録されていることを示し，0 の時はされていないことを示す．この flag は GC 実行時の表の検索に使用する．これら参照表内の二つの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．

なお，本提案手法において想定するコールスタックの構造は**図 8** に示すとおりであ

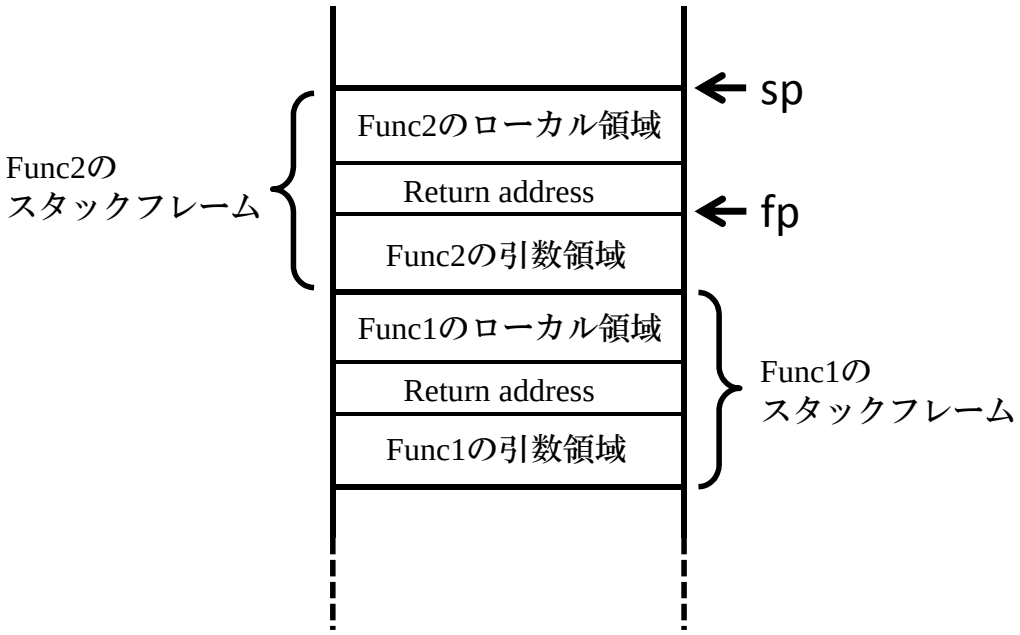

図 8: コールスタックの構造

るとする．図 8 は，サブルーチン Func1 から呼び出されたサブルーチン Func2 が実行中である時のコールスタックの様子を表している．簡単のため，スタックフレームにはそのルーチンの局所変数領域と引数領域，呼び出し元へのリターンアドレスが格納されるとする．また，sp と fp はそれぞれスタックのトップとリターンアドレスが格納されている場所を指すとする．スタック最上部，つまり実行中のルーチンのフレームをカレントフレームと呼ぶ．

次に，表に対する操作が発生する場面とその際の動作について述べる．まず，サブルーチンがコールされると，スタックへそのルーチンのフレームがプッシュされる．この時，コールされたサブルーチンへの引数が新たにスタックに格納されるため，引数に含まれるポインタを表に登録する．この登録の動作例を**図 9** に示す．スタックフレームの引数領域に格納されたポインタは，まず一時登録表に登録される (a)．この時，参照表に直接登録しないのは，引数を格納している段階では，FrameID に使用する fp の値がまだ呼び出し元のフレームを指しているためである．そして引数を格納後，リターンアドレスが格納され fp の値が更新された時点で，一時登録表に登録されているエントリを参照表に登録する．この時，登録されたエントリが持つ flag の値を 1 にセットする．

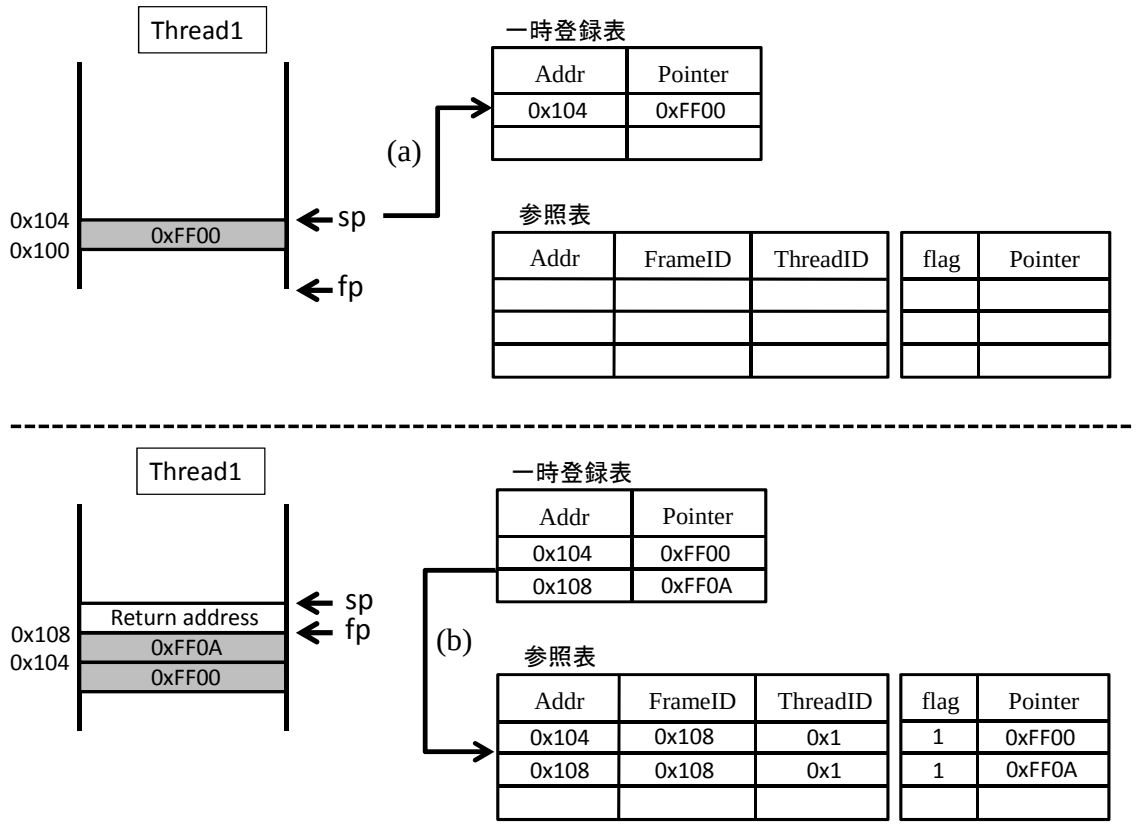

図 9: 登録時の動作

この他に登録が発生する状況としては，カレントフレームに新たな値が格納されるといった場面がある．これは，サブルーチン内で新たに局所変数を宣言する際に発生する．この変数がポインタならば参照表へ登録する．なお，この場合は既に fp の値は更新済であるので一時登録表への登録は必要ない．また，登録の他に既に登録されているポインタの値が変更されることも考えられる．この場合は，変更されるポインタが格納されているスタック上のアドレスで Addr を検索し，値が一致したエントリの Pointer の値を変更する．

次にエントリの削除が発生する場面について述べる．エントリの削除が発生する場面は大きく分けて，フレームポップ時と，スレッドの終了に伴ってコールスタックが破棄される時の二つである．まず，フレームポップ時にはそのフレーム内に格納されていたポインタに対応するエントリをすべて削除しなければならない．そこで，このようなエントリを検索するために FrameID を用いる．先述した通り，FrameID の値はエントリ登録時に fp が指しているアドレスであるため，FrameID はスタックフレーム

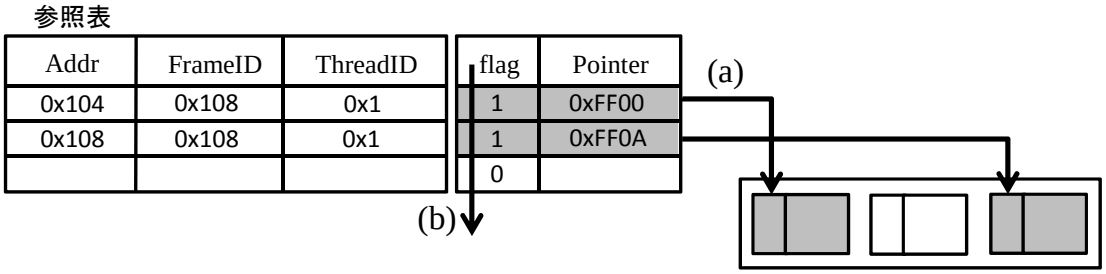

図 10: 参照表を使用したマーキング

毎に一意に決定される．よって，FrameID の値とポップするスタックフレームの fp が指しているアドレスが同じ値であるエントリを検索することで，ポップするスタックフレームに含まれるエントリを特定し，削除することが可能である．次に，コールスタックの破棄時には，そのコールスタックに格納されていたポインタが登録されているエントリをすべて削除しなければならない．このようなエントリを検索する場合は ThreadID を用いる．具体的には，破棄するスレッドのスレッド ID と等しい ThreadID を持つエントリを検索し削除する．この時，削除されたエントリは flag の値を 0 にリセットする．

この様にスタックに対する操作に対応して表を操作することで，スタック内に格納されているヒープ領域中のオブジェクトへのポインタのみを表に登録することができる．よって GC 実行時にコールスタックを走査せずとも，参照表に登録されているポインタを辿ることで，生きているオブジェクトへマークすることができる．**図 10** に参照表を使用したマーキングの動作例を示す．flag の値が 1 であるエントリの Pointer に格納されているポインタを辿ってオブジェクトにマークする (a)．これを全てのエントリに対して行うことで，コールスタックに格納されているポインタが指すオブジェクトに対する全てのマークが完了する (b)．

以上の様に，表を用いてコールスタックに格納されているポインタを管理することで，GC 実行時のコールスタックの走査を省略でき，それまでその処理にかかっていたサイクル数を削減できる．

4.3 実装において考慮すべき点

前節で述べた提案手法を実装する上で考慮すべき点が二つ存在する．本節ではそれらの点について述べる．まず考慮しなければならないのが，登録ポインタ数が表のエ

ントリ数を越えた場合の動作である．表のエントリ数は有限であるため，保持できるポインタの数には限りがある．そのため，コールスタックの状態によっては表の空きがなくなり，新しくポインタを登録できないという状況が発生すると予想される．この様な表溢れによって登録できないポインタがある場合，GC 実行時にそのポインタが指すオブジェクトのマーク漏れが発生し，生きているオブジェクトが誤って回収されてしまう．そのため表が溢れてしまった場合は表の利用を停止し，従来通り，コールスタックを走査する様にする必要がある．なお，表の利用の再開は，通常のコールスタックの走査と並行してポインタを表に再登録することで可能であると考えられる．しかし，表の利用を再開した後も表溢れが頻発するようであれば登録のコストが増大してしまうことも考えられる．そのため，表の利用の再開のタイミングについては，実際のアプリケーションの動作を調査した上でのさらなる考察が必要である．また，表溢れの発生率と，表を拡大するためハードウェアコストを考慮して，表のサイズを設定することも必要となる．

もう一つ考慮すべきなのが，ポインタと非ポインタをどの様にして判別するかということである．スタックフレーム内にはサブルーチン内のローカル変数や引数の値が含まれるが，これらはポインタと非ポインタの識別なく格納されている．こうした値はただのビット列に過ぎず，どれが表に登録すべきヒープ領域中のオブジェクトへのポインタか区別できない．このことは通常の Mark & Sweep でも問題となるが，アルゴリズムの実装におけるこの問題の解決策として，ある程度の精度でポインタを判別するという方法がある．この方法では，ポインタの値が持つべきいくつかの条件，例えば，0 でない，ヒープ領域内を指している，正しくアライメントされた値である等，を満たしているかどうかで値がポインタであるかを判別する．しかしこの方法では，生きているオブジェクトを破棄しないという GC の原則は守られるが，非ポインタが誤ってポインタと判別された結果，本来ゴミとして破棄されるはずのオブジェクトが回収されずにヒープ領域上に残ってしまうことがある．そのため，この方法はメモリリソースが限られている組込みシステムにはメモリ使用効率の点から適さない．そのため，スタックに対するフレームのプッシュやポップ，カレントフレーム内でのポインタの生成等に伴って表を操作する様な ABI(Application Binary Interface) を新たに設計する必要があると考えられる．

5　提案手法の適用による効果の考察

前章では，表を用いてポインタを管理することでコールスタック走査を高速化する手法を提案した．本提案手法を適用することで，従来コールスタック走査にかかっていたサイクル数を削減できるが，その一方で，表の操作にかかるオーバーヘッドが発生してしまうと考えられる．そこで，提案手法によって削減が期待されるサイクル数と発生するであろうオーバヘッドの両方を考慮し，提案手法を適用した際の効果について考察する．

5.1　表の操作によるオーバヘッドの見積もり

4.2 節で述べた通り，表に対する操作が発生するのは以下の時である．

- スタックフレームへの引数格納時 (登録)
- カレントフレーム内でのローカル変数格納時 (登録)
- カレントフレーム内でのポインタの更新時 (更新)
- スタックフレームポップ時 (削除)
- スレッド終了に伴うコールスタック破棄時 (削除)

そこで，これらの回数を，表に対して操作が行われる回数に換算し，操作のオーバーヘッドを見積もる．

評価環境は 3.1 節で述べた予備評価と同様である．KVM 上で実行するプログラムである Minibenchmark を分析し，引数格納回数，ローカル変数格納回数，ポインタの更新回数を算出する．引数格納回数は，各メソッドの引数に含まれるオブジェクトへの参照の数と，そのメソッドの呼び出し回数から算出した．同様にして，ローカル変数格納回数は，各メソッド内でのローカル変数の宣言数と，そのメソッドの呼び出し回数から算出した．なお，Java をはじめとするオブジェクト指向言語ではメソッドがコールされると，this ポインタと呼ばれるそのメソッド自身のオブジェクトへの参照がスタックに格納される．この this ポインタが格納されるのは fp が更新された後であるので，これもローカル変数に含まれる参照の数としてカウントする．this ポインタの格納回数はメソッドの総呼び出し回数，つまりコールスタックへのフレームプッシュ回数と等しいとする．KVM においてフレームのプッシュは pushFrame 関数として実装されているので，この関数の呼び出し回数を this ポインタの格納回数とする．また，ポインタの更新回数はメソッド中でオブジェクトに対する参照が書き換えられている箇所の数と，そのメソッドの呼び出し回数から算出した．なお，削除回数は登録回数と

表 2: 表に対する操作の回数

引数格納時の登録回数	47560 回
ローカル変数格納時の登録回数	72390 回
ポインタ更新回数	11 回
削除回数	119950 回

表 3: 操作の想定コスト

引数格納時の登録	6 サイクル
ローカル変数格納時の登録	4 サイクル
ポインタの更新	2 サイクル
エントリの削除	2 サイクル

等しいものと仮定している．

5.2 評価結果

前節で述べた方法で算出した各操作の回数を**表 2** に示す．引数格納の回数が 47560 回，ローカル変数の格納回数が 72390 回であった．このうち this ポインタの格納回数は 42632 回であった．また，ポインタの更新回数は 11 回であった．削除回数は登録回数と等しいと仮定しているので，引数格納回数とローカル変数格納回数を合計した値の 119950 回となった．

この結果から，表の操作にかかるオーバーヘッドを見積もる．各操作にかかるコストは**表 3** に示すとおりであるとする．なお，ここではプロセッサと CAM が同等のクロック周波数で動作していると仮定している．そのため，更新，削除の際に必要となるエントリの検索には 1 サイクルを要するとする．また，参照表及び一時登録表への書き込みは 1 ワードあたり 1 サイクルを要すると仮定した．表 2 と表 3 から，提案手法の適用によって発生する表の操作コストは 814842 サイクルであると見積もれる．

5.3 考察

提案手法ではコールスタックに格納されているヒープ領域への参照を保持する表を導入するが，この表は，KVM の GC アルゴリズムにおける getGCRegisterMask 関数内で作成されるスタックマップに相当するものである．よって，提案手法を適用することでスタックマップを作成する必要がなくなり，この関数の実行に要していたサイクル数を削

減できると考えられる．3 章で行った予備評価では，getGCRegisterMask 関数は約 427 万サイクルを要していた．一方，前節では提案手法の適用による表の操作のオーバーヘッドを 814842 サイクルであると見積った．このオーバヘッドは getGCRegisterMask 関数の実行サイクル数と比較して約 1/5 である．したがって，提案手法を適用することによって，従来 getGCRegisterMask 関数に要していた実行サイクル数を大きく削減できると考えられる．

ここで，提案手法の適用による消費エネルギー効率への影響について考察する．提案手法を適用することで実行サイクル数が大きく削減され，それに伴い消費エネルギーの削減が期待できる．しかし一方で，表の追加によってハードウェア量が増え，消費電力が増加してしまうことも予測される．特に，参照表の一部を構成する CAM は連想検索が可能である一方，その消費電力が大きいことが知られている．提案手法適用前のシステムのサイクル当たりの消費電力及び実行サイクル数をそれぞれ P と T，ハードウェア追加によって増加するサイクル当たりの消費電力を P'，提案手法によって削減されるサイクル数を t と置くと，この提案手法によって削減される消費エネルギー E は

$$E = P \cdot t - P' \cdot (T - t)$$

という式で表される．この E の値が正の時，提案手法によってシステム全体の消費エネルギー効率が向上していることが示される．よって，E の値が最大となる様な提案手法の実装方法を検討していくことが本研究の今後の重要課題である．

その他に考えられる提案手法の効果として，GC によるプログラムの最大停止時間の短縮が挙げられる．KVM で実装されている GC アルゴリズムは，GC の実行中はミューテータの実行を停止させている．しかし，本研究が対象とする組込みシステム，特にモバイル端末には，ユーザとインタラクティブにやり取りすることを前提としたアプリケーションが多く存在する．この様なアプリケーションでは GC による最大停止時間の短さが求められるが，通常 GC のスループットの高さと最大停止時間の短さはトレードオフの関係にある．しかし，提案手法では GC の実行サイクル数そのものが削減されることが予測されるので，スループットを犠牲にすることなくプログラムの最大停止時間を短縮することができると考えられる．

6　おわりに

本論文では，既存の GC の動作を分析しアルゴリズムのボトルネックを調査した．その結果，ルート集合，特にコールスタックの走査に多くのサイクル数を要していることが分かった．この結果をもとに，コールスタック内のポインタを表に記憶しておくことでコールスタックの走査を高速化する手法を提案した．また，提案手法がもたらす効果について考察した．ポインタを記憶しておく表の操作にかかるコストを見積もり，提案手法によって削減されると思われるサイクル数と比較した．その結果，提案手法を適用することで多くのサイクル数の削減が見込めることが分かった．

本研究の今後の課題として，今回高速化の対象としたルート集合の走査とは別の，負荷が大きい処理に対するハードウェア支援手法を検討する，また各アルゴリズムとモバイル端末における他の代表的アプリケーションの組み合わせにおいて，多くの実行サイクル数を要する箇所を調査し，その処理に適したハードウェア支援手法を提案していく，といったことが挙げられる．また，今回提案した高速化手法の実装も今後行っていく．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わり，幾度となく貴重な助言を頂いた名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩准教授に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室，齋藤研究室，及び松井研究室の先輩，同期，そして研究グループ内の方々に深く感謝致します．特に研究に関して貴重な意見を下さった辻孝辰氏，江藤正道氏，安井裕亮氏，神村和敬氏に感謝致します．ありがとうございました．

参考文献

[1] McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, Part I, *Commun. ACM*, Vol. 3, pp. 184–195 (1960).

[2] Minsky, M.: A LISP Garbage Collector Algorithm Using Serial Secondary Storage, Technical report, Cambridge, MA, USA (1963).

[3] Collins, G. E.: A method for overlapping and erasure of lists, *Commun. 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] Takeuchi, I., Yamazaki, K., Amagai, Y. and Yoshida, M.: Lisp can be "Hard" Real Time.

[6] Click, C., Tene, G. and Wolf, M.: The pauseless GC algorithm, *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, New York, NY, USA, ACM, pp. 46–56 (2005).

[7] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A. and Werner, B.: Simics: A Full System Simulation Platform, *Computer*, Vol. 35, No. 2, pp. 50–58 (2002).