卒業研究論文

ハードウェア支援によるコールスタック上のポインタ判別を用いた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 21 年度入学　21115008 番

井手上 慶

平成 25 年 2 月 12 日

ハードウェア支援によるコールスタック上のポインタ判別を用いたGC高速化手法の検討

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，これらの組み込みシステムに対してより複雑で多岐に渡る機能が求められるようになり，その性能要求が増大してきている．こうした性能要求への対応として高クロックなプロセッサの利用が考えられるが，これでは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．

さて，汎用計算機と比較した場合，これらの機器では小容量のメモリしか搭載できないためメモリ管理の重要性がより高く，特にガベージコレクション (Garbage Collection: GC) の働きが重要となる．この 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 の高速化は主にソフトウェア面で古くから研究されてきているにも関わらず，未だ根本的な問題解決には至っていない．

そこで本論文では，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，機器の性能向上と低消費電力を両立するプロセッサの実現を目標とする．

本論文では，GC アルゴリズムの基本的な構成処理要素である，ルートからのポインタ探索に着目し，これを高速化する手法を提案する．ルートとは，メモリ上に確保されたオブジェクトへの参照の起点となる集合である．ルートの一要素であるコールスタック上には，オブジェクトへのポインタだけでなく，int 型変数等の数値も含まれている．そのため，GC 実行時にメモリ上のオブジェクトを探索する際には，この中のポインタのみを識別する処理が必要となる．そこで，コールスタック上に格納されているポインタのみを予め表に記憶しておくことで，ポインタの判別処理を削減する手法を提案する．これにより、従来ポインタ判別に要していたサイクル数を削減し、GC の高速化を図る．

提案手法の効果について，SPECjvm2008 ベンチマークを用いて評価した．表の操作にかかるコスト，および提案手法により削減できるサイクル数を見積もった結果，多くのベンチマークプログラムにおいて GC の高速化が見込めることがわかった．

ハードウェア支援によるコールスタック上のポインタ判別を用いたGC高速化手法の検討

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，これらの組み込みシステムに対してより複雑で多岐に渡る機能が求められるようになり，その性能要求が増大してきている．現状，各メーカーではこの性能要求に応えるべく，1GHz を超えるクロック周波数で動作するプロセッサを利用している．しかし，これはモバイルプロセッサとしては高クロックであり，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となっている．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，プロセッサの性能向上によりクロック周波数を相対的に低く抑えることができれば，さらなる低消費電力化にも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要因の一つに，**ガベージコレクション (Garbage Collection: GC)** が挙げられる．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．そのため，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考えられる．一方，GC の高速化に関する研究は古くから行われているが，その多くはソフトウェア面からのアプローチであり，GC 実行時のプロセス全停止によるレスポンス低下等の問題があることから，根本的解決と呼べるものではない．また，ハードウェア面からの高速化については，これまであまり議論されてこなかった．

そこで本論文では，GC の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，機器の性能向上と低消費電力を両立するプロセッサの実現を目標とする．そのために，まずは多くの GC アルゴリズムにおける基本的な構成処理要素を調査する．そしてその結果を基に，ハードウェア支援による GC 高速化手法を提案する．そして提案手法による効果を見積り，手法の有効性を確認する．

以下，2 章では研究背景として，組込みシステム，GC，および既存のハードウェア支援による GC 高速化研究について述べる．3 章では多くの GC アルゴリズムの基本的な構成処理要素を調査し，4 章では予備評価として，その処理のために必要となるコストを測定する．5 章では着目した構成処理要素に代わるハードウェア支援手法を

提案し，6 章で提案手法の効果を見積もる．そして 7 章で結論を述べる．

2　研究背景

本章では組込みシステム，GC，既存のハードウェア支援手法について述べる．

2.1　組み込みシステム

現在，テレビ，エアコンといった家電製品や，自動車，航空機といった輸送機器など，様々な機器が計算機によって制御されている．このような機器に組み込まれ，特定の機能を実現する計算機システムを**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，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，組み込みシステムに対してより複雑で多岐に渡る機能が求められるようになり，その性能要求は増大してきている．さらにこれらのモバイル機器においては，単に高い処理性能だけでなく，省電力化等によるバッテリ持続時間の向上も求められている．

このようなモバイル機器では，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ため，メモリ管理を行う GC の重要性が非常に高く，システムに与える影響も大きい．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であるウェブブラウザ上の JavaScript エンジンでは，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，モバイル機器に対する性能要求に応えるためには，プロセッサの高性能化だけでなく，この GC の高速化も重要となる．さらに，GC を高速化できれば，機器の性能向上だけでなく，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と考えられる．

2.2　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，C や C++を除くほとんどの言語処理系に実装されており，重要な機能の一つとなっている．

ここで，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．なお，ヒープ領域と

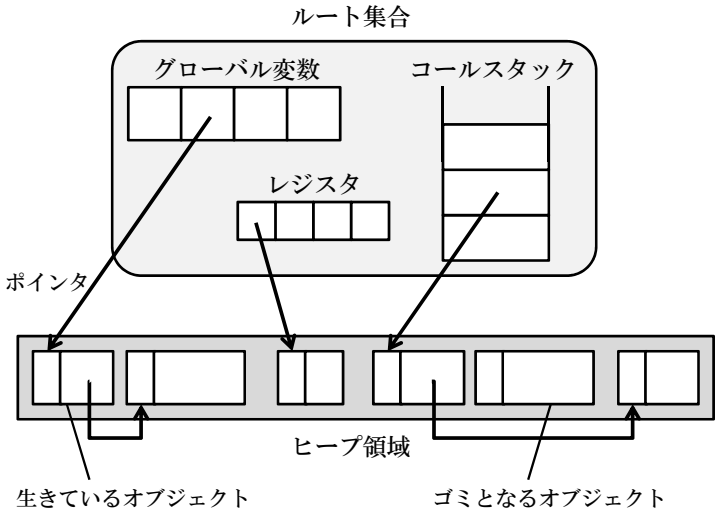


図 1: プログラム実行時のメモリの様子

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，メモリ確保の際にはこのヒープ領域からオブジェクト単位で領域を確保する．なお，このようにメモリ領域の確保を必要とするアプリケーション等のプログラムのことを**ミューテータ**という．

また，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，レジスタ等の，ミューテータ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，ヒープ領域中のオブジェクトはルート集合を起点として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領域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．

なお，ミューテータはオブジェクトの生成に伴うメモリ領域の確保だけでなく，ポインタの書き換えを行うことでオブジェクト間の参照関係も変化させる．その結果，ルート集合から参照されなくなるオブジェクトが発生する可能性があり，これらは二度と参照されない不要な領域となってしまう．そのため，そのようなオブジェクトは GC 実行時にゴミとして判断され，割り当てられていたメモリ領域が解放される．

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，

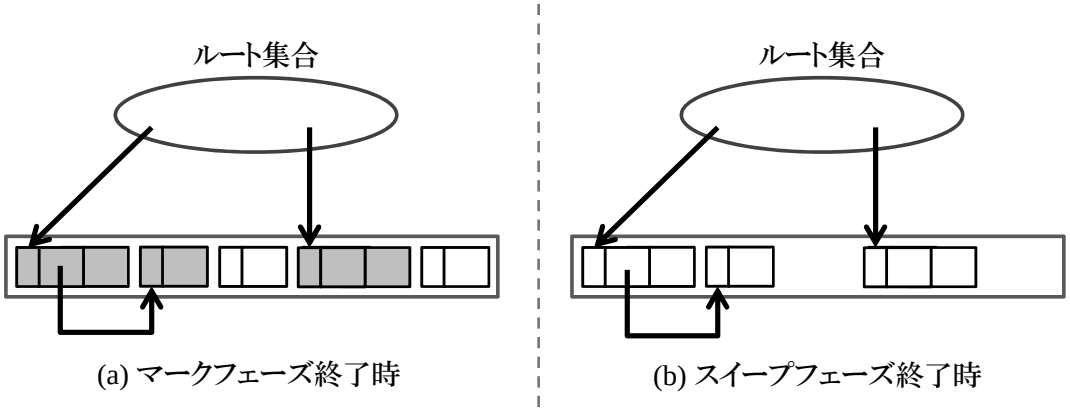


図 2: Mark & Sweep の動作例

Mark & Sweep [1]，Copying [2]，Reference Counting [3] という，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 [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，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 2 つのフェーズで構成される．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．なお，オブジェクトへのマークは，各オブジェクトが持つヘッダの中にマークビットを用意しそれを用いることで実現する．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)．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であると判断できる．そこでマークフェーズが終わると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を解放する (b)．

このアルゴリズムのメリットとして，アルゴリズムがシンプルであるため実装が比較的容易であることが挙げられる．しかしデメリットとして，スイープフェーズでは不要な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ことが挙げられる．

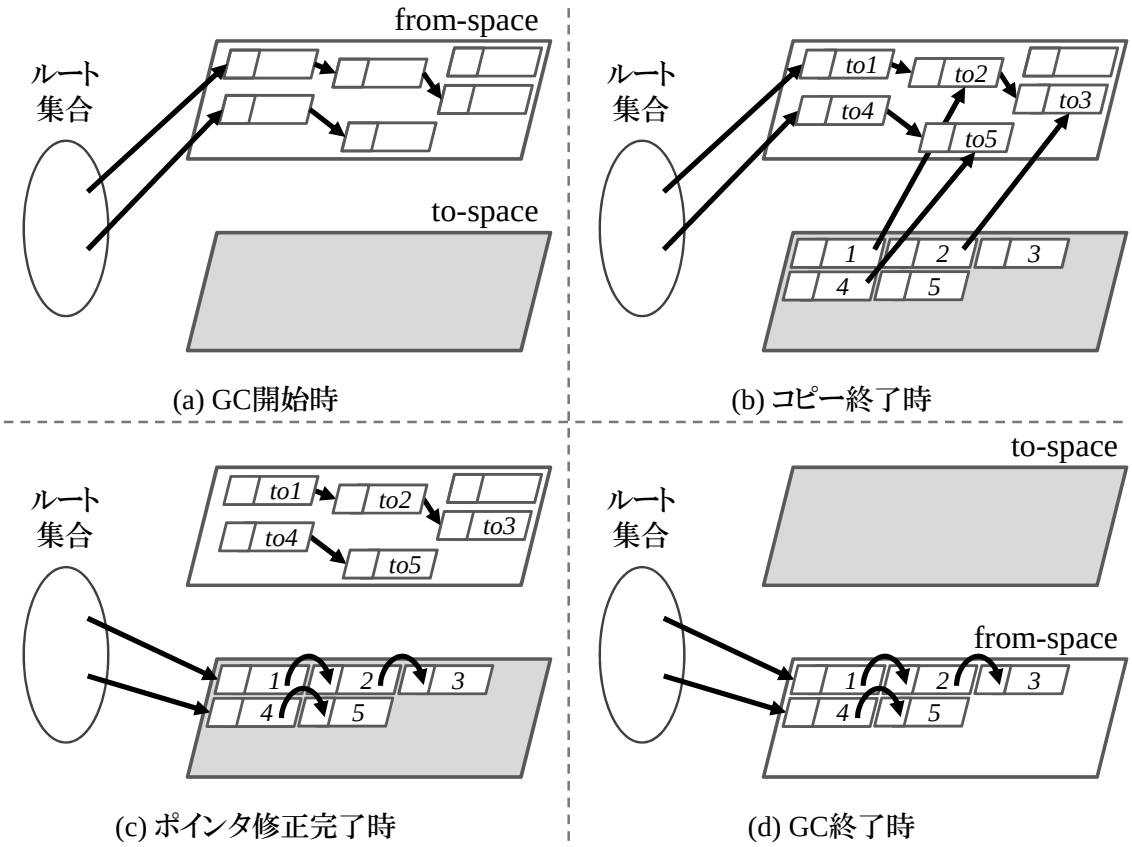


図 3: Copying の動作例

2.2.2 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，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．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 (b) では，ルート集合や to-space 上の各オブジェクトが持つポインタはまだ from-space を指しているため，そのようなポインタを to-space 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ため， from-space 全体を解放し，to-space と from-space を入れ替える (d) ．このように，Copying では 2 つの領域を入れ替えなが

ら交互に使用する．

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，同時にヒープ領域のコンパクションが可能となる点が挙げられる．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 全体を解放するだけであるため，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ため，ヒープ領域の使用効率が悪いというデメリットがある．

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．この参照カウンタは，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メモリ領域を解放する．

Reference Counting 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 の実行時に初めてメモリ不足の原因となる不要な領域を解放するが，Reference Conting では不要な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，ポインタの更新を行うミューテータの実行に伴ってゴミが生成される度に，ゴミとなったオブジェクトのみを回収するため，ミューテータの最大停止時間が非常に短くなることも挙げられる．

しかしデメリットとして，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点が挙げられる．また，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メモリ領域の使用効率を悪化させてしまう．

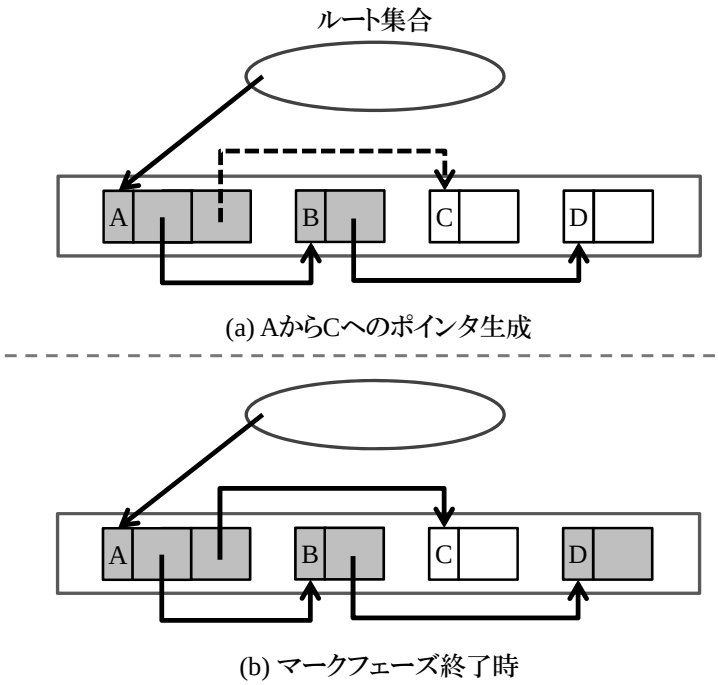


図 4: ポインタの書き換えによるマーク漏れ

2.3 既存のハードウェア支援手法

前節で述べた 3 つの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．本節ではその中でも代表的な二つの既存研究を取り上げ，こ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 [5] 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 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は，それぞれ別のプロセスとして並行に動作している．このように，ミューテータと並行に動作する GC を Concurrent GC[6] と呼ぶ．

しかし，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へのマーク漏れが発生する可能性がある．**図 4** にこの問題が発生する場合の動作例を示す．GC プロセス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，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．この時，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 (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，その書き込みによるデータ不整合を防ぐために行う同期処理のことである．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よう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，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．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）[7] 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 3 つのフェーズからなる．そして，NAP はこれら 3 つのフェーズが全て，ミューテータである Java スレッドと並行して動作する Concurrent GC である．そのため，GC スレッドとミューテータ間で何らかの同期処理が必要となる．NAP ではこの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ために行う同期処理のことである．

マークフェーズをミューテータと並行して動作させた場合，SILENT と同様，生きているオブジェクトへのマーク漏れという問題が起こり得る．そこで NAP では，GC スレッドが巡回したオブジェクトにマークすると同時に，巡回したポインタにもマークをすることでこれを解決している．Java ではオブジェクトが 8 バイト境界に配置されるため，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する．これを Not Marked Through(NMT)-bit と呼ぶ．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，リードバリアによりそのポインタの NMT-bit をチェックする．もし GC スレッドが未巡回であれば 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．NMT-bit のチェックに要するオーバヘッドは 1 サイクル程度と，通常のロード命令と比較して小さいため，非常に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のページを探し，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ミューテータと並行して実行することは非常に難しい．しかしNAPでは，リードバリアを用いた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，すぐには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．このように，ページに特殊な保護をかけることで，保護されたページ内のオブジェクトの読み出しを検知可能とし，リードバリアによる同期処理を実現している．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．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，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．なお，リードバリアを用いてNMT-bitをチェックする手法は従来から提唱されていたが，ソフトウェア面における改良では高負荷な処理となることからこれまで実現はされてこなかった．これをハードウェア支援による高速なリードバリアを用いて実現したのがNAPである．しかしこのリードバリアは，Concurrent GCにおけるGCプロセスとミューテータ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，得られる効果は限定的であると考えられる．また，Pauseless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．そのため，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．

3　コールスタック上のポインタ判別

2.3 節で述べた既存手法は，いずれも GC プロセスとミューテータが並行して動作する Concurrent GC を対象としている．この Concurrent GC は，一部の mark/sweep 操作のオーバラップ等によって，GC 実行に伴うシステムの最大停止時間を緩和する手法である．しかし，並行動作によって GC 処理のスループットが犠牲となっており，GC の根本的な問題解決と呼べるものではない．そして SILENT や NAP は，この Concurrent GC において必要となる，GC プロセスとミューテータ間の同期処理を高速化しているに過ぎない．そこで本論文では，基本的な GC アルゴリズムの実行時間の多くを占めている構成処理要素に着目し，この処理を高速化することで根本的な GC 処理の高速化を目指す．本章では，着目する構成処理要素，およびこの処理を実現するためのソフトウェア面での既存手法について順に述べる．

3.1　ルートからのポインタ探索

本論文では，多くの GC アルゴリズムにおいて，ルートからポインタをたどるという共通の動作が必要となる点に着目する．2.2 節で述べたとおり，ルートとは，アプリケーションからヒープ領域内のオブジェクトへの参照の集合である．Mark & Sweep 等では，GC 実行時にこのルート内のポインタをたどる処理が必要となるが，この際，ルートの一要素であるコールスタックの走査には時間がかかることが知られている．その理由として，コールスタック上に積まれた値が，ヒープ領域中のオブジェクトを指しているポインタかどうかを判別する必要があることが挙げられる．コールスタック上に積まれているコールフレームには，関数内のローカル変数や引数などが格納されている．このローカル変数には，int や float などのプリミティブ型変数とオブジェクトへの参照を示す参照型変数があり，これらは区別無くコールスタック上に含まれている．そのため，GC 実行時にはこれらを区別するための処理が必要となる．現在使用されている GC アルゴリズムは，これらの区別を厳密に行うかどうかで保守的 GC と正確な GC の二つに大別できる．

3.1.1　保守的 GC

保守的 GC とは，プリミティブ型と参照型の厳密な区別を行わない GC アルゴリズムである．しかし，単純にコールスタック上の全ての値を参照型変数とみなした場合，ポインタと誤識別される非ポインタが大量に発生してしまう．そのため，保守的 GC

ではいくつかの条件にあてはまるかどうかでポインタの識別を行う．

例えばCPUのアラインメントを利用した識別が挙げられる．ポインタの値は，32ビットCPUの場合には4の倍数，64ビットCPUの場合には8の倍数になるため，これと異なる場合は非ポインタであると識別できる．その他の識別条件としては，そのポインタがヒープ領域内を指しているかどうか，あるいはポインタがオブジェクトの先頭を指しているかどうか等が挙げられる．

保守的GCのメリットとして，GC実行時にプリミティブ型と参照型の厳密な区別を行う必要がないため，言語処理系の実装が比較的容易であるという点が挙げられる．しかしデメリットとして，非ポインタをポインタと誤識別した場合，それが本来回収されるはずのオブジェクトであっても生きていると見なしてしまうため回収することができず，ヒープ領域を圧迫してしまう点が挙げられる．また，コピーGCなどのようなポインタの書き換えを含むGCアルゴリズムの場合，本来はint型などの非ポインタの値を書き換えてしまう可能性があるなど，使用されるGCアルゴリズムが制限されてしまう点も挙げられる．

3.1.2　正確なGC

正確なGCとは，プリミティブ型と参照型の厳密な区別を行うGCアルゴリズムである．この区別のためには言語処理系に何らかの工夫が必要となる．例えば，非ポインタ全てに，ポインタと区別するためのタグを付けておくという手法がある．タグ付けの方法は様々であるが，一例として下位1ビットをタグとして用いる方法が挙げられる．CPUのアラインメントにより，ポインタ値の下位1ビットは必ず0となる．そこで非ポインタの値に対し，値を左に1ビットシフトした後，下位1ビットのビットを立てることで，タグを付けることが可能となる．しかし，この方法ではシフト演算によるオーバーフローを考慮する必要がある．また，計算の際にはタグを取り外し，計算結果に再度タグを付けなくてはならない．

このタグ付けのように，正確なGCの実現に必要となる処理に要するコストは大きいことが予想される．しかし，保守的GCのような誤識別の問題や，GCアルゴリズムが制限されるということはない．そのため，厳密な区別のためのコストを削減し，GCアルゴリズムに制限のないこの正確なGCを高速化できれば，より多くのGCアルゴリズムの高速化に繋がると考えられる．

3.2　スタックマップ

3.1.2項で述べた正確なGCを実現する手法の一つに，JavaVMの一種であるHotspot-

```
1 class Sampleprogram{
2     public static void main(String args[ ]){
3         int val = 1;
4         Object ref = new Object();
5     }
6 }
7
```

図 5: サンプルプログラム

```
1 pc( 0): iconst_1
2 pc( 1): istore_1
3 pc( 2): new
4 pc( 5): dup
5 pc( 6): invokespecial
6 pc( 9): astore_2
7 pc(10): return
```

図 6: 変換後のバイトコード

VM で用いられているスタックマップというものがある．Java コールスタック上の各フレームは，ローカル変数配列と，スタックマシンである HotspotVM が計算の際に利用するオペランドスタックを持っている．スタックマップはこの中の参照型変数が格納されている位置を示すものである．

HotspotVM は，バイトコード内の命令セットを一つずつ読み出して実行するインタプリタ方式を採用しており，スタックマップは，これらのバイトコードに含まれる型情報に応じて作成される．例えば，**図 5** に示すプログラムを実行する場合を考える．このプログラムは，二つのローカル変数 val と ref に対し，それぞれ int 型の値と参照型の値を格納するものである．このプログラムはまず，**図 6** に示すバイトコードへと変換される．図 6 は，メソッド内のバイトコードに一意に割り振られているプログラムカウンタ (pc) と，命令のオペコードを人間が読める形式で表現したニーモニックを表している．この中で，ローカル変数へ値の格納を行うバイトコードは 2 行目の istore_1 と，6 行目の astore_2 であり，それぞれローカル変数に対して，int 型の値，および参照型の値を格納するものである．図 6 の場合，istore_1 は 1 つ目のローカル変数 val に int 型の値 1 を，astore_2 は 2 つ目のローカル変数 ref に Object というクラスのインスタンスのアドレスを格納している．この istore や astore のように，バイトコードには型情報を含んでいるものがある．そこで，この型情報を用いることで，ローカル変数配列内で参照型変数が格納されている位置を特定することが可能となるため，スタックマップを作成することができる．

なお，スタックマップの作成は，抽象的インタプリタでバイトコードを実行することで行っている．抽象的インタプリタとは，通常のインタプリタとは異なり，バイトコードの型情報のみを記録するインタプリタである．図 6 で示したバイトコードに対

表 1: 抽象的インタプリタの動作例

バイトコード	抽象的インタプリタ実行フロー
pc(0): iconst_1	pc(0): locals='r**', stack=''
pc(1): istore_1	pc(1): locals='r**', stack='v'
pc(2): new	pc(2): locals='rv*', stack=''
pc(5): dup	pc(5): locals='rv*', stack='r'
pc(6): invokespecial	pc(6): locals='rv*', stack='rr'
pc(9): astore_2	pc(9): locals='rv*', stack='r'
pc(10): return	pc(10): locals='rvr', stack=''

する抽象的インタプリタの実行フローの様子を表 1 に示す．表 1 中の locals はローカル変数配列，stack はオペランドスタックを表しており，その中の r(reference) は参照型，v(value) はプリミティブ型を意味している．また，locals 内の*はまだ初期化されていないローカル変数を示している．なお，メソッドに与えられた引数もローカル変数に含まれる．そのため，各プログラムカウンタにおいて，ローカル変数の先頭は参照型である引数 args を表す r が記録されている．抽象的インタプリタは，あるバイトコード実行前のローカル変数とオペランドスタックの型情報を記録する．例えば，pc(2) では直前に実行された istore によって，ローカル変数 val に int 型の値 1 が格納されるため，locals には先頭から，引数である参照型変数 args を表す r と，プリミティブ型変数 val を表す v が記録されている．

このように，抽象的インタプリタは実際の値などは考慮せず，型情報のみを記録していく．そしてこの抽象的インタプリタが記録した型情報をもとに，フレーム内のローカル変数およびオペランドスタックの中で参照型変数が格納されている位置のビットを立てたビット列を生成する．例えば，pc(9) の astore_2 実行後のビット列生成の様子を**図 7** に示す．この時点でのローカル変数配列は，先頭から，メソッドに与えられた引数 args の参照型の値 (a)，istore_1 によって変数 val に格納された int 型の値 (b)，astore_2 によって変数 ref に格納された参照型の値 (c) となっており，101 というビット列が生成される (d)．この様に作成されたビット列をスタックマップとして利用することで，GC 実行時にコールスタック上に存在するポインタのみを正確に判別することが可能となる．しかし，条件分岐等によってプログラムの実行フローが変化する可能性もあるため，各実行フローに合わせたスタックマップが必要となる．しかし，全て

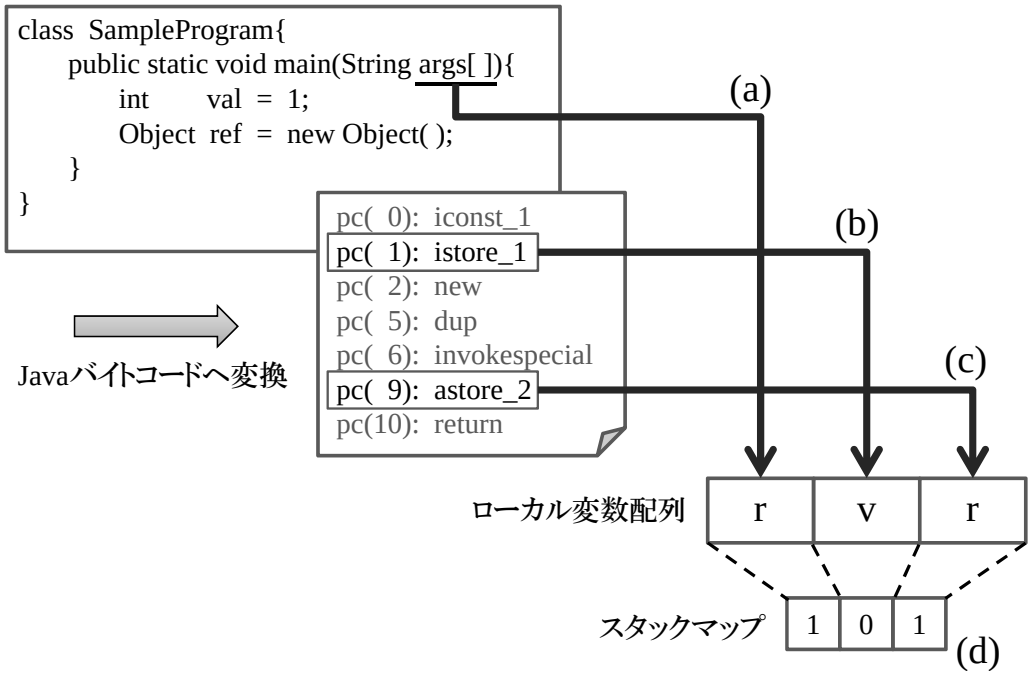


図 7: スタックマップ作成の様子

の実行フローに対応するスタックマップを作成し，これを保持しておくことはメモリの使用効率を悪化させてしまう．そのため，スタックマップは GC 実行時に適宜作成される必要があり，そのコストは大きいことが予想される．

4　スタックマップ作成コストの測定

次に，予備評価として 3.2 節で述べたスタックマップの作成に要するコストを測定する．これにより，スタックマップ作成部分に，どの程度性能改善の余地があるか調査する．なお，HotspotVM において抽象的インタプリタを呼び出し，スタックマップを作成している関数は compute_map 関数である．そのため，この compute_map 関数の実行に要するサイクル数をスタックマップ作成に要するコストとして測定する．

4.1　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 [8] を用いた．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る ARM アーキテクチャを選択した．ARMv7 は，32 ビットの RISC マイクロプロセッサ，ARM-RealView PBX は，ARMv7 を搭載するシステム開発用ベースボードである．そして，シミュレート実行するアプリケーションとして HotspotVM を使用する．なお，

表 2: シミュレーション対象となるプロセッサの構成

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

評価には SPECjvm2008 ベンチマークを使用し，これを HotspotVM 上で実行してサイクル数を測定する．

ここで，HotspotVM の GC アルゴリズムについて述べる．HotspotVM では世代別 GC というアルゴリズムを採用している．世代別 GC とは，オブジェクトを複数の世代に分類し，その世代毎に適した GC アルゴリズムを適用するというものである．HotspotVM の世代別 GC では，生成されて間もない新世代オブジェクトと，生成後一定の期間が経過した旧世代オブジェクトの二世代にオブジェクトを分類する．そして新世代オブジェクトに対しては Copying を適用し，旧世代オブジェクトに対しては Mark & Sweep を改良した Mark Compact[9] という GC アルゴリズムを適用している． Mark Compact は，Mark Sweep の短所の一つであるフラグメンテーションを回避するため，マークフェーズ終了後に生きているオブジェクトを移動させるコンパクションフェーズを実行し，ヒープ領域のコンパクションを行うものである．

4.2 評価結果

スタックマップを作成する compute_map 関数のサイクル数は，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の値が到達した時点の CPU サイクルをシミュレータから取得することで算出する．評価の結果を**図 8** に示す．グラフの横軸は使用したベンチマーク名，縦軸は GC 全体の処理に要したサイクル数を 100%とした場合の，compute_map 関数の実行サイクル数の割合を示している．例えば scimark.fft.small の場合，GC 全体の処理に対する compute_map 関数の実行サイクル数の割合は約 15%であることを示している．

評価結果を見ると，最も compute_map 関数の割合が大きいのは crypto.rsa であり，その割合は約 18.2%と GC 実行時間の多くを占めていることがわかる．逆に最も割合が小さいのは crypto.aes であり，その割合は約 1.5%であった．全てのベンチマークで平均すると，スタックマップ作成にかかるサイクル数は，GC 全体の処理の約 9.4%で

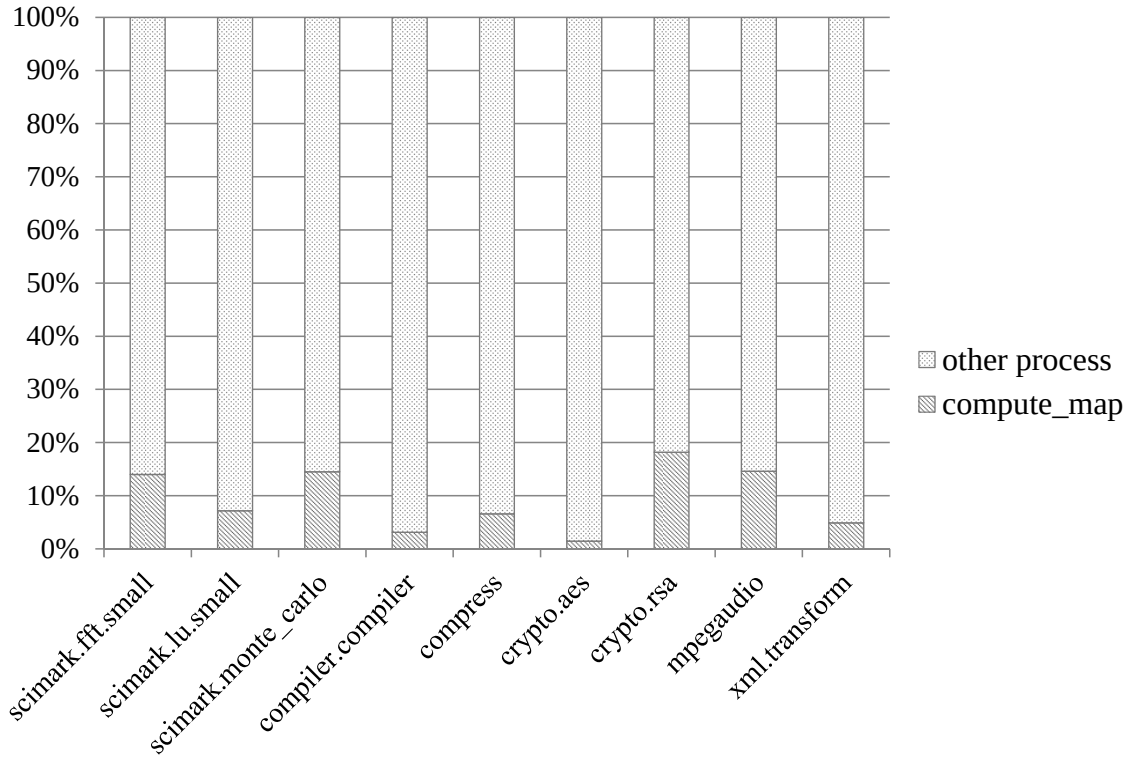


図 8: スタックマップ作成コストの測定結果

あり，この処理が GC の性能に与える影響は大きく，GC の性能を大きく改善できる可能性があることを確認できた．

5　ハードウェア支援による GC 高速化手法

前章での予備評価により，スタックマップ作成のコストは大きく，GC 全体の処理に与える影響も大きいことを確認できた．そこで本章では，このスタックマップを用いることなく，GC の高速実行を可能にするハードウェア支援手法を提案する．本論文で提案する手法は，ポインタを管理する新たな表を，連想検索，およびシーケンシャルアクセスが可能なバッファメモリを用いて実現することで，GC 実行時のポインタ判別を支援するというものである．表を用いてポインタのみを管理することにより，従来必要であったポインタ判別の処理が不要となり，GC の高速化に繋がると考えられる．

なお，本論文で実装対象とする HotspotVM は，組み込みシステムで広く用いられている ARM アーキテクチャもサポートしていることに加え，多くの処理系で用いられている Mark & Sweep を改良した Mark Compact を採用している．そのため，本論文で提案する手法の適用範囲は広いと考えられる．

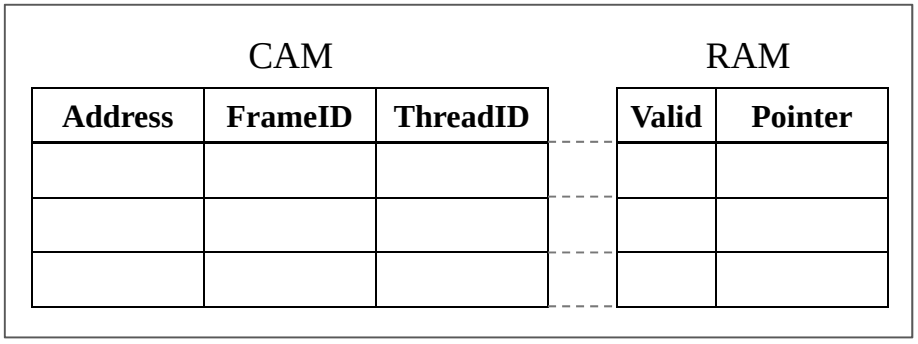


図 9: ポインタ管理表の構成図

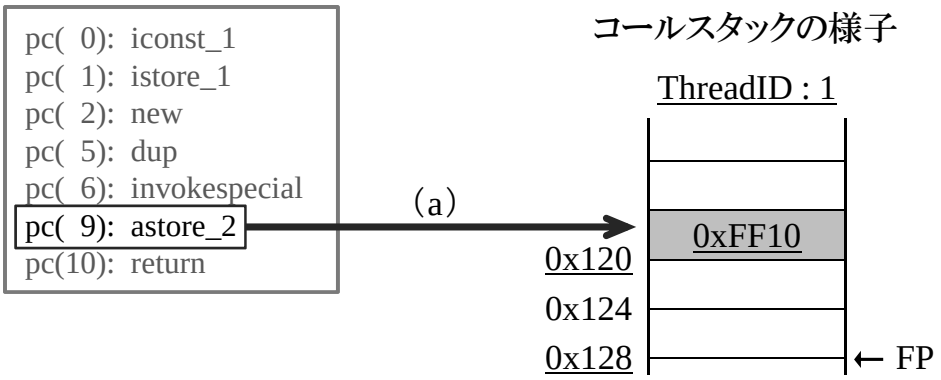
次節からは，提案するポインタ管理表の構成とその動作，および表のエントリが溢れた場合の処理について順に述べる．なお，本章で述べる表に対する操作は，命令セットアーキテクチャを拡張し，これを利用するための GC ライブラリの実装により実現することを想定している．

5.1 ポインタ管理表の構成

新たに追加するポインタ管理表は，高速な連想検索が可能な汎用 3 値 CAM（Content Addressable Memory）と，シーケンシャルアクセスが可能な RAM を組み合わせた実装を想定している．追加するポインタ管理表の構成を**図 9** に示す．このポインタ管理表は二つの表で構成されており，左側の表はエントリの識別情報を格納するために，右側の表は GC 実行時に必要とされる情報を格納するために使用する．なお，こ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．

このうち左側の表は，ポインタが格納されているコールスタック上のアドレスを示す Address，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 の三つのフィールドを持つ．なお，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の値を登録する．これらの各フィールドの値は，ポインタの登録や更新，削除の際に，その操作の対象となるエントリを特定するために用いる．そこで，高速な連想検索が可能な CAM で表を実装することにより，エントリの特定にかかるオーバヘッドを低く抑えることが可能となる．

また，右側の表は，そのエントリが持つポインタが有効なオブジェクトを指しているかどうかを示すフラグを格納する Valid と，コールスタック上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．Valid の値が 1 の時



(b) 登録エントリの決定

Address	FrameID	ThreadID	Valid	Pointer
0x108	0x114	0x01	0	0xFE60
0x134	0x140	0x02	1	0xF190

(c) ポインタの登録

Address	FrameID	ThreadID	Valid	Pointer
0x120	0x128	0x01	1	0xFF10
0x134	0x140	0x02	1	0xF190

図 10: ポインタ登録時の動作

は登録されているポインタが有効であることを示し，0 の時は有効でないことを示す．よって，ポインタを削除する場合にはこの Valid を 0 にすればよい．これらのフィールドは，GC 実行時に参照され，表に登録されている各エントリが持つポインタの有効性を判断し，有効なポインタのみを辿るために使用する．GC 実行時には表の全エントリを走査することになるため，これらのフィールドはシーケンシャルにアクセス可能な RAM によって実装する．また，ある特定のエントリに対する操作の際には，まず Address などの値を用いて表の CAM を検索しインデクスを得ることで，RAM で構成された表の対応するエントリに対しても高速にアクセス可能となる．

5.2 ポインタの登録時の動作

ポインタ管理表へのポインタの登録は，3.2 節で述べたスタックマップの作成を参考に，実行するバイトコードに含まれる型情報に応じて行う．この登録時の動作例を**図 10** に示す．この例では，astore 命令によって，コールスタック上 0x120 番地のロー

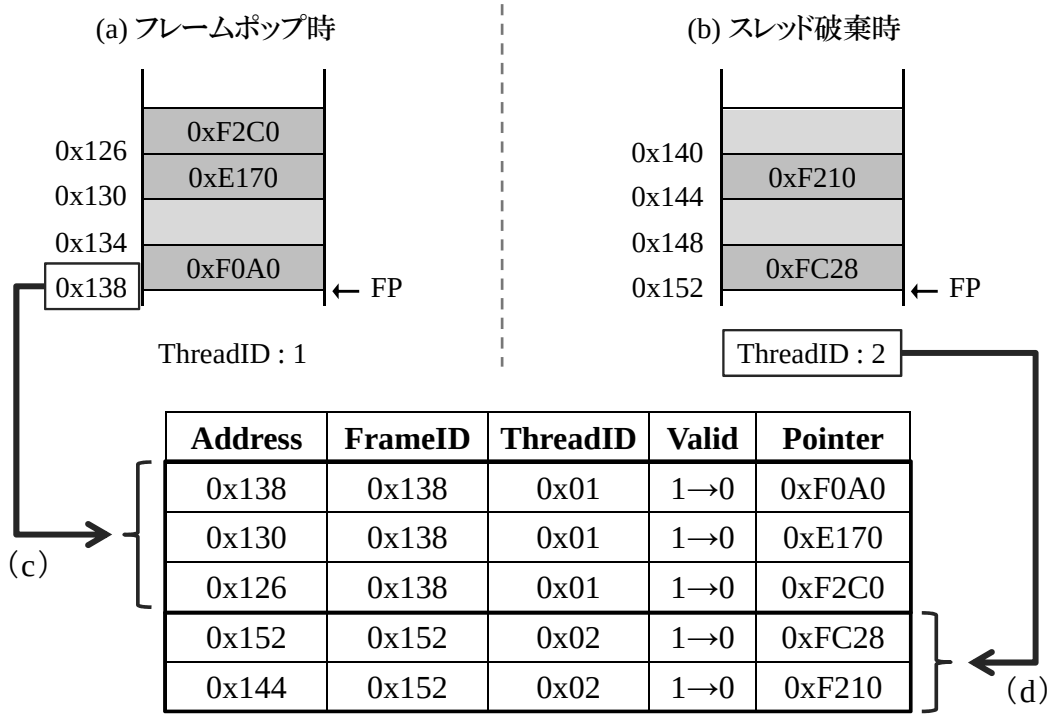
カル変数に 0xFF10 というポインタを格納している (a)．提案手法では，この astore 命令を実行する際に，表へポインタを登録する処理を加える．まず Address，FrameID，ThreadID を用いた連想検索により，すでに同じローカル変数に対する登録が行われているかどうか検査する．これにより，同一のローカル変数に対応するエントリが複数登録されないようにする．図 10(b) に示すような表の場合，登録する 0x120 番地のローカル変数に対応するエントリはまだ登録されていない．そこで，次に表の各エントリの Valid を参照し，有効なポインタを保持していない，つまり Valid の値が 0 であるエントリを検索することで，ポインタを登録するエントリを決定し，各フィールドに値を登録する (c)．図 10 の場合，Address の値は，ポインタを格納しているローカル変数領域のアドレスである 0x120 となる．同様に，FrameID には実行時のフレームポインタの値である 0x128，ThreadID には自身のスレッド ID である 1 を登録し，Pointer に 0xFF10 を登録する．そして登録時には Valid を 1 にセットすることで，ポインタが有効であることを示しておく．

5.3 ポインタの更新，削除時の動作

表に対する操作には，ポインタの登録だけでなく更新や削除も必要となる．これらの操作は，登録時と同じように Address，FrameID，ThreadID を用いた連想検索によりエントリを特定して行う．

ポインタの更新は，コールスタック上の同一のローカル変数に対して新たなポインタが格納された場合に発生する．この場合，通常の登録時と同様の連想検索によりエントリを特定する．これにより，更新するローカル変数に対応するエントリを一意に決定することが可能である．エントリを特定したらその Pointer を更新し，Valid を 1 にセットしておく．このような動作により，あるローカル変数に対する更新が可能となる．しかし，この更新処理によって Pointer に NULL ポインタが登録される場合も考えられる．その場合は対象エントリに格納されたポインタの削除と見なし，Valid を 0 にセットすることでエントリが有効でないことを示しておく．

なおポインタの削除に関しては，複数エントリに対する操作も必要になる．例えば、**図 11**(a) に示す状況を考える．この図は，ある 3 つのポインタを格納しているフレームがポップされる直前のコールスタックの様子を表している．このフレームをポップする場合，フレーム内の 3 つのポインタに対応するエントリは全て無効にする必要がある．そこで FrameID，および ThreadID を用いた連想検索により，同一スレッド内の同じ FrameID を持つエントリを特定する．先述したとおり，FrameID はエントリ登



Address	FrameID	ThreadID	Valid	Pointer
0x138	0x138	0x01	1→0	0xF0A0
0x130	0x138	0x01	1→0	0xE170
0x126	0x138	0x01	1→0	0xF2C0
0x152	0x152	0x02	1→0	0xFC28
0x144	0x152	0x02	1→0	0xF210

図 11: ポインタの削除時の動作例

録時のフレームポインタの値である．そのため，フレームをポップする直前のフレームポインタの値を用いることで，ポップされるフレーム内の全てのポインタに対応するエントリの特定が可能となる．図 11(a) の場合には，その時点のフレームポインタの値である 0x138，および自身のスレッド ID である 1 を用いた連想検索によって，削除されるポインタに対応するエントリを特定できる (c)．そして該当する全てのエントリの Valid を全て 0 にセットしておくことでそのエントリが有効でないことを示しておく．

また，あるスレッドの終了に伴ってコールスタックが破棄される場合にも，複数エントリに対する削除が発生する．例えば図 11(b) の様なコールスタックを持つスレッドが破棄される場合を考える．この例の場合，コールスタック上には 2 つのポインタが存在しており，これらは全てエントリから削除する必要がある．このようなエントリは ThreadID を用いた連想検索により特定する．この例の場合，破棄するスレッドの ThreadID は 2 であるため，これと等しい ThreadID を持つエントリを検索する (d)．そして，該当するエントリのフラグを全て 0 にセットしておくことで，エントリが有効でないことを示しておく．

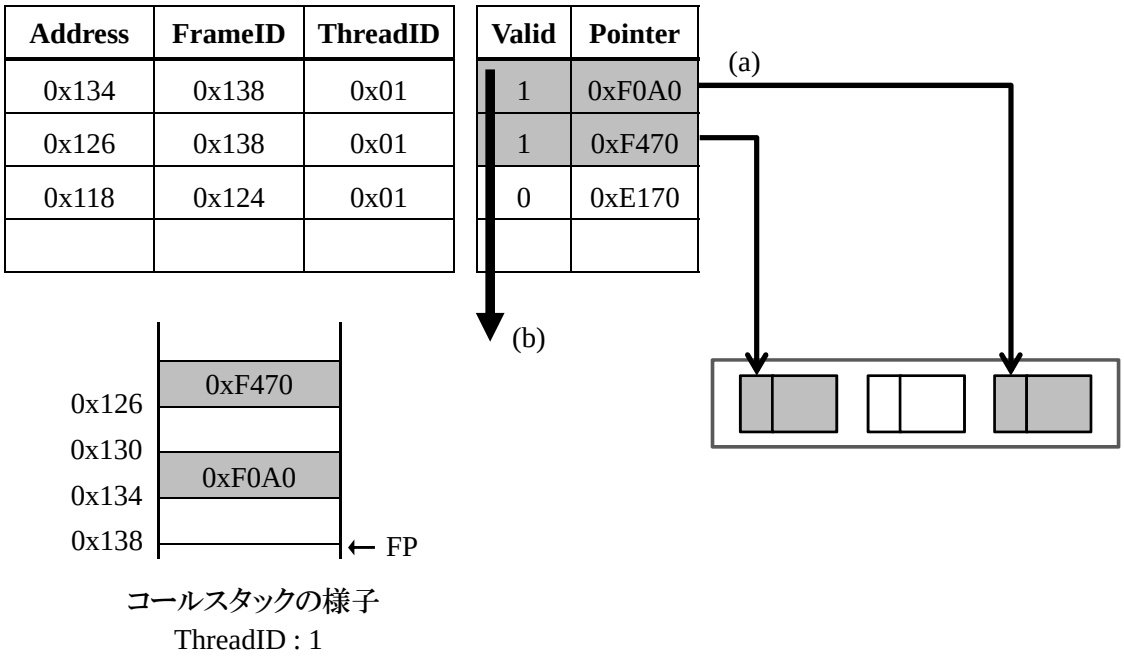


図 12: GC 実行時の動作例

5.4 GC 実行時の動作

前節までに述べた操作によって表に登録されたポインタは，GC 実行時にルートとして利用する．表にはコールスタック上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が全て登録されているため，この表を参照することでコールスタック上のポインタ判別を行うことなく，生きているオブジェクトへのポインタ探索を開始できる．なお GC 実行時に参照するのは，表のうち RAM で構成された部分のみである．その動作例，および GC 実行時のコールスタックの様子を**図 12** に示す．この例は，コールスタック上のあるフレーム内に 2 つのポインタが格納されており，これらが表に登録されている状態で GC が実行された場合を表している．

GC 実行時にはまず，表のエントリを 1 つずつ参照していき，Valid の値が 1 であるエントリを検索する．図 12 の表の場合，コールスタック上の 0x134 番地，および 0x126 番地に格納されたポインタがこれに該当する．そのため，これらに対応するエントリの Pointer に格納されている，0xF0A0，および 0xF470 が指しているオブジェクトをマークする (a)．そして，表の全エントリを参照し，Valid が 1 であるポインタを全て辿ることで，コールスタック上に存在するポインタが指すオブジェクトを全てマークすることができる (b)．以上の動作を GC 実行時に行うことで，これまではコールスタック上の各フレームが持つローカル変数に対し，スタックマップを用いて行ってい

たポインタ判別の処理を削減できると考えられる．

5.5 表が溢れた場合の処理

これまでに述べた提案手法を実装する上で考慮すべき点として，登録ポインタ数が表のエントリ数を越えた場合の処理が挙げられる．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可能性がある．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には表の利用を停止し，従来通り，スタックマップを用いた処理に切り替える必要がある．

HotspotVM ではコールスタックに積まれた各フレームに対し，必要に応じて compute_map 関数を呼び出すだけでスタックマップを作成することが可能である．そのため，処理を切り替える際に特別な操作は必要なく，従来通りコールスタック上の各フレームに対してスタックマップを計算することで，これを利用してポインタを辿ることができる．

6 提案手法による効果の見積り

本章では，提案手法により得られる効果について考察する．提案手法では，HotspotVM が用いているスタックマップを必要としないため，これを作成する compute_map 関数の実行に要するサイクル数を削減できると考えられる．しかし，追加した表へのアクセスに伴うオーバヘッドが発生するため，これを見積もる必要がある．次節からは，このオーバヘッドを見積もり，提案手法によって削減できると予想されるサイクル数と比較することで，提案手法の効果について考察する．

6.1 表へのアクセスに伴うオーバヘッド

オーバヘッドの見積りは，表を構成している CAM と RAM のそれぞれに対するアクセスレイテンシ，および表へのアクセス数から計算できる．そこで，まず表へのアクセスレイテンシを見積もる．なお，提案手法で追加する表のエントリ数は 4K 行とする．また，使用する CAM は，MOSAID 社の DC18288[10] を参考にし，その動作周波数を 200MHz と想定する．CAM の連想検索は 1 サイクルで可能であるが，シミュレート対象の CPU の動作周波数は 2GHz であるため，これを周波数比を考慮して CPU サ

表 3: 表へのアクセスに必要なコスト

CAM へのアクセス	10cycle
RAM へのアクセス	2cycle
1 エントリの登録	12cycle
1 エントリの参照	2cycle

イクルに換算すると 10 サイクルと考えることができる．また，RAM はシミュレート対象のマシンにおける L1 キャッシュと同等のサイズを用意することで，4K 行の CAM と同数のエントリを確保できる．そこで RAM 構成部の 1 エントリに対するアクセスを，L1 キャッシュのアクセスレイテンシと同等の 2 サイクルと仮定する．

以上のレイテンシを用いて，表へのアクセスに必要となるコストを見積もった結果を表 3 に示す．ポインタ登録時などの特定の 1 エントリに対する操作のコストは，エントリの特定に必要な CAM へのアクセスに 10 サイクル，エントリ特定後の RAM へのアクセスに 2 サイクル必要となり，合計で 12 サイクルであると考えられる．また，GC 実行時の表の 1 エントリに対する参照には，RAM へのアクセスに必要な 2 サイクル必要となる．これらのコストを用いて，提案手法におけるオーバヘッドを算出する．

6.2 削減サイクル数との比較

前節で述べたコストを用いてオーバヘッドを算出するために，表へのアクセス回数を計測した．なお，この計測は 4 章で述べた予備評価と同様の環境で行った．計測結果を表 4 に示す．ここで，登録回数とは，表へのポインタの登録を行う astore 命令の実行数，参照回数は GC 実行時に参照されたエントリ数の合計を表している．この計測結果に，前節で述べた表へのアクセスに必要なコストを乗じたものを提案手法におけるオーバヘッドとし，削減できると予想されるサイクル数と比較して提案手法の効果を見積もる．

各ベンチマークプログラムにおける見積り結果を**図 13** に示す．図中では，各ベンチマークプログラムの評価結果が 3 本のグラフで表されており，左から，スタックマップを用いた既存手法の GC 実行サイクル数，既存手法から compute_map 関数のサイクル数を除いたサイクル数，オーバヘッドを考慮した提案手法の実行サイクル数を表しており，既存手法を 1 として正規化している．

この結果を見ると，scimark.fft.small，scimark.lu.small，scimark.monte_carlo，com-

表 4: 表へのアクセス回数の計測結果

	登録回数	参照回数
scimark.fft.small	249,437	152,000
scimark.lu.small	257,964	152,000
scimark.monte_carlo	241,534	24000
compiler.compiler	8,212,363	312,000
compress	296,178	52,000
crypto.aes	16,653,149	384,000
crypto.rsa	17,461,968	84,000
mpegaudio	566,414	164,000
xml.transform	9,148,177	648,000

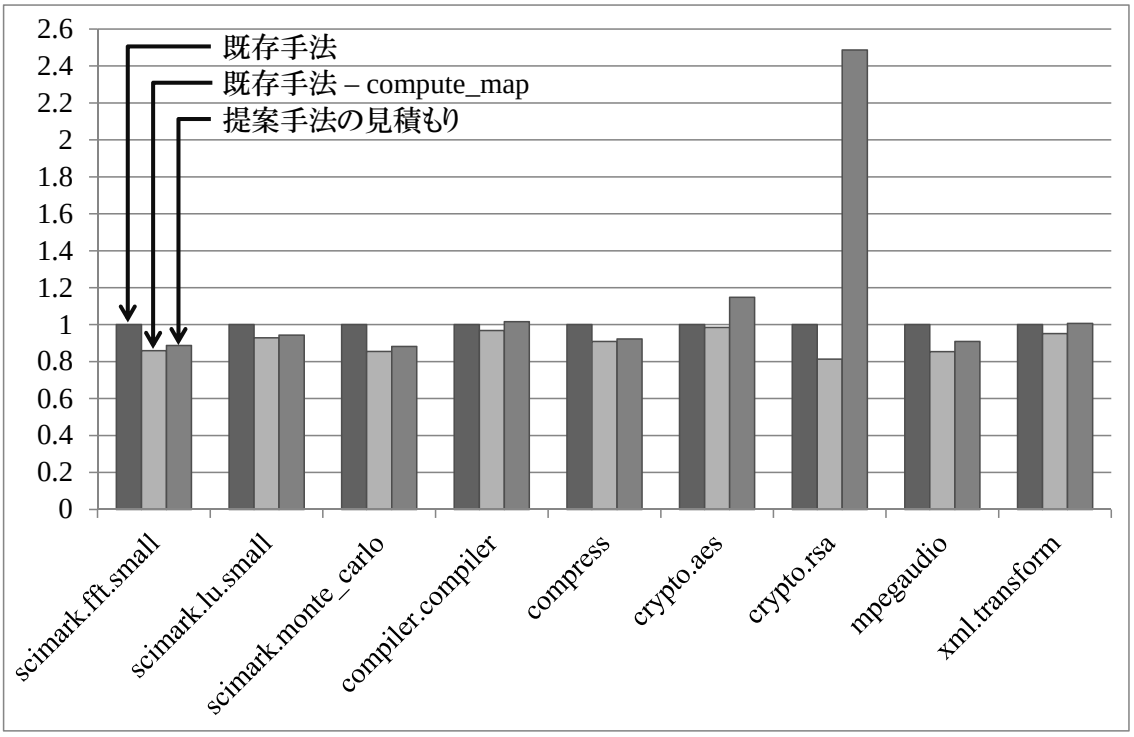


図 13: 提案手法による効果の見積り

press，mpegaudio の 5 つのベンチマークプログラムにおいて，表の操作によるオーバヘッドを考慮した場合でも，GC 実行サイクル数を削減できていることが確認できる．特に scimark.monte_carlo の場合，提案手法では既存手法に比べて GC 実行サイクル数を約 11.8%削減できており，提案手法による効果が大きいことがわかる．

しかし，残りのベンチマークプログラムでは既存手法よりも実行サイクル数が増加している．compiler.compiler と xml.transform ではその増加量はわずかであるが，crypto.aes，crypto.rsa では大きく増加してしまっている．特に crypto.rsa では実行サイクル数が既存手法の約 2.5 倍となっており，オーバヘッドが非常に大きいことがわかる．この原因として，crypto.rsa の場合，ベンチマークプログラムにおける GC 実行サイクル数が，ベンチマークプログラム全体の実行サイクル数と比較して 0.15%と非常に少ないということが挙げられる．そのため，表へのポインタの登録が増えてオーバヘッドが増加する一方，削減できるサイクル数が相対的に少なくなり，提案手法による効果があまり得られなかったと考えられる．

このようなプログラムに関しては，GC 実行回数に対して，提案手法を適用するかどうかを決定するための閾値を設定し，GC 実行数が閾値を越える場合，つまり提案手法による効果が見込める場合にのみ手法を適用するという解決方法が考えられる．これにより，実行するプログラムに応じて処理を切り替えることで，提案手法による実行サイクル数の増加を防ぐことが可能となる．このような改良を施すことで，様々なプログラムに対して提案手法による GC の高速化が期待できると考えられる．

7　おわりに

本論文では，多くの GC アルゴリズムの高速化を目的として，GC の基本的な構成処理要素であるルートからのポインタ探索に着目し，これを高速化する手法を提案した．ルートの一つであるコールスタック上のローカル変数には，ポインタだけでなく int などの数値も含まれているため，GC 実行時にはこの中のポインタのみを判別するための処理が必要となる．そこでこのポインタ判別を支援して GC を高速化するために，コールスタック上のポインタのみを表に記録しておくことで，ポインタ判別処理を削減する手法を提案した．また，提案手法による削減サイクル数，および表の操作にかかるコストを見積り，提案手法の効果について考察した．その結果，提案手法を適用することで GC の高速化が見込めることがわかった．

本研究の今後の課題として，ポインタ管理表のハードウェアコスト，およびその操作にかかるコストの削減が挙げられる．例えば，ポインタ管理表の分割による GC 実行時の表の参照コストの削減などが考えられる．また，提案手法ではハードウェアを拡張しているため，これに伴って必要となる消費エネルギーコストを調査し，低消費電力を実現するための手法を提案していく．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，梶岡慎輔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里見優樹氏，山崎一樹氏，福田諭氏に感謝致します．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] Ossia, Y., Ben-Yitzhak, O., Goft, I., Kolodner, E. K., Leikehman, V. and Owshanko, A.: A parallel, incremental and concurrent GC for servers, *SIGPLAN Not.*, Vol. 37, No. 5, pp. 129–140 (2002).

[7] Click, C., Tene, G. and Wolf, M.: The pauseless GC algorithm, *Proc. 1st ACM/USENIX international conference on Virtual execution environments*.

[8] 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, *SIGARCH Comput. Archit. News*, Vol. 39, No. 2, pp. 1–7 (2011).

[9] Haddon, B. K. and Waite, W. M.: A Compaction Procedure for Variable-Length Storage Elements, *The Computer Journal*, Vol. 10, No. 2, pp. 162–165 (1967).

[10] MOSAID Technologies Inc.: *Feature Sheet: MOSAID Class-IC DC18288*, 1.3 edi-

tion (2003).