高いヒープ使用率の下で高速なインクリメンタルGC

白井達也[†] 遠藤敏夫[†]
田浦健次朗[†] 近山隆[†]

ガーベジコレクション（GC）の性能は一般にスループットと停止時間で表される．本稿では停止時間が短く，ヒープ使用率の増加にともなうスループットの低下を抑えたGC手法を述べる．この目的のために，incremental mark sweep（IMS）とreference counting（RC）を組み合わせたうえでIMS処理量の動的な制御方法を提案する．IMSとRCを単純に組み合わせただけでは必要以上に頻繁にMSサイクルを行ってしまう．そこでIMSのマーク量を動的に制御することが重要になる．本稿ではコストのモデルを用いてIMSを動的に制御することによるスループットの向上の効果を述べる．また提案したGCをJikes RVMに実装し，停止時間が短くヒープ使用率が高くてもスループットの低下を抑えられることを示す．

Fast Incremental GC under High Heap Residency

Tatsuya Shirai,[†] Toshio Endo,[†] Kenjiro Taura[†]
and Takashi Chikayama[†]

The performance of garbage collectors (GC) is generally evaluated by its throughput and pause time. We propose an incremental collector scheme that suppresses the throughput decrease caused by increase in heap residensy. To achieve this, we design a hybrid collector that combines Incremental Mark-Sweep (IMS) and Reference Counting, then propose a dynamic control method of IMS. Improvement on collection throughput of the proposed dynamic IMS control scheme was shown through theoretical cost analysis. We implemented proposed hybrid GC sheme on Jikes RVM and compared with other GC algorithms. We show proposed collector suppresses the decrease in throughput while it keeps short pause time.

1. はじめに

近年ではプログラムの規模が増大し，それにともないプログラム中のデータ構造が非常に複雑になっている．そのような状況ではメモリの解放をプログラマが明示するのは非常に困難であるため，自動的に動的なメモリ管理を行うガーベジコレクタ（GC）[4]が重要になっており，すでに多くの言語処理系で実装されている．

GCの性能は「処理時間に対するメモリの解放量（スループット）」と「ユーザプログラムを連続して停止させた時間（停止時間）」，そして「ヒープサイズの低下にともなうスループットの低下の程度（ヒープサイズの変化への耐性）」という指標で評価される．数値計算のようなプログラムでは停止時間は重要ではないため高いスループットだけが求められる．一方，ゲームのようなインタラクティブなアプリケーションで停止時間が長くなると，アプリケーションの反応時間が長くなりユーザにストレスを与えてしまう．そのため，スループットが良いだけでなく停止時間が短いことが重要になる．

ヒープサイズの変化への耐性はGC方式によって異なる．停止型トレースGCの一種であるmark sweep（MS）方式ではヒープサイズを小さくするとスループットは大きく下がる．さらに，MSの停止時間を短くしたincremental mark sweep（IMS）方式では，この傾向は顕著になる．Reference counting（RC）は他の方式よりヒープサイズの影響が少ない方式だが，それでも循環構造のごみを解放する場合はヒープサイズの低下によりスループットが下がる．

この3つの指標がすべて重要になるアプリケーションとしては，たとえばゲームサーバや携帯端末があげられる．ゲームサーバや携帯端末で用いられるような対話的アプリケーションでは，停止時間が長ければユーザが対話的に操作を行うことができない．さらに

† 東京大学
The University of Tokyo

ゲームサーバなどでは処理できるクライアント数の限界はネットワーク性能や計算速度に加え，サーバが使用可能なメモリ量によっても決定される．このような場合，ヒープサイズの数分の 1 しかアプリケーションが使用できない状況は望ましくない．つまり，ヒープ使用率を高めたいという要求がある．また，携帯端末のような組み込めるメモリのサイズが小さい環境では，やはりアプリケーションが使用するメモリサイズよりはるかにヒープサイズを大きくすることは困難である．以上のような場合には，ヒープ使用率が高くても良好なスループットを得ることが重要になる．

以上の目的のために，本研究では RC を基本とし，循環ごみの解放のために IMS を組み合わせる GC を構築する．RC と IMS を単純に組み合わせた場合，必要以上に多く GC 処理を行ってしまい，スループットが低下してしまう．そのため我々は IMS の処理量を適切に動的制御する手法を提案する．これにより，ヒープ使用率が高い場合に他方式よりもスループットを良くすることができることを示す．

本稿の構成は以下のようになる．まず 2 章で関連研究を述べる．3 章ではメモリ割当てとマーク量のモデルを述べ，IMS における GC コストを示す．4 章ではマーク量の動的制御の必要性を述べ，提案する GC アルゴリズムを説明する．5 章でモデルをもとにコストを見積もり，各方式の比較を行う．6 章で実験結果を述べる．最後に 7 章で本稿のまとめと今後の課題を述べる．

2. 関 連 研 究

本章では本研究の基礎となる基本的な GC アルゴリズムについて述べ，さらに本研究とは違うアプローチで GC の性能を向上させる研究にも触れる．

2.1 Reference Counting

Reference counting（RC）は，他のオブジェクトからの参照の数（参照回数）を維持し，参照回数が 0 になったらそのオブジェクトを解放するという手法である．この手法はユーザによる参照の更新のたびに参照回数の増減を処理するため，停止時間が短く，オブジェクトがごみになると同時に解放することができるという特徴を持つ．また，この手法は後述の mark sweep GC などと異なり，GC コストはヒープサイズの影響を受けない．一方 RC の問題の 1 つは，そのコストは多くの場合にトレース GC よりも大きい傾向にあることである．ただし，後述のようにそれを緩和する方法がすでに提案されている．もう 1 つの問題は，循環参照構造のグラフを成すようなオブジェクトは，どのオブジェクトも参照回数が 0 にならないため，その全体がごみとなっても解放することができないということである．そのため，cycle collection を組み合わせる必要がある．

2.1.1 Deferred Reference Counting

ユーザによる参照の更新のうち大部分はヒープのオブジェクトからの参照ではなく，スタックやレジスタなどのルートからの参照である．Deferred reference counting [5] はルートからの参照を数えないことで，参照回数の維持のコストを大幅に減らすアルゴリズムである．ユーザプログラムによる更新の際にデクリメントに関してはデクリメントされるオブジェクトのアドレスをバッファ（デクリメントバッファ）にためる．そして定期的にルートをスキャンして，ルートから直接参照されているオブジェクトをマークする．そのあとバッファ内のオブジェクトをデクリメントして，参照回数が 0 になり，かつマークされていなかったら解放する．このアルゴリズムではオブジェクトがごみになった瞬間に解放することはできないが，ルートをスキャンする頻度を高くすればごみになってから解放するまでの時間は十分に短くすることができる．また，ルートをスキャンする時間も十分短いため停止時間は短いままである．

以下，本稿ではこの手法を取り入れたものを単に RC と呼ぶ．

2.1.2 Cycle Collection

循環ごみを解放する方法は以下の 2 つに分かれる．

1 つ目はときどきバックアップ GC としてトレース GC を用い，循環参照を解放する手法である．この手法ではバックアップ GC による停止時間も考慮する必要があるため，停止時間を短くするためには，バックアップ GC としてもインクリメンタルなものを採用する必要がある．DeTreville はバックアップ GC として IMS を用いた実装を行った [2] が，その性能の定量的な評価までは報告されていない．

2 つ目は参照グラフ内の参照を一時的に参照回数から除くことにより循環ごみを見つける，cycle detection を用いた手法である．この手法ではデクリメントされたが解放されないオブジェクトを循環ごみの候補と見なし，そのようなオブジェクトを起点として循環参照を調べる．さらにデクリメントされた後にすぐにそのオブジェクトを調べるのではなく，バッファにためることでコストを大きく減少させることができる [6]．たとえばメモリが不足したらバッファ内のオブジェクトの何割かについて循環参照を調べるといった方法が考えられる．しかし，上記の方法では停止時間が長く

なってしまう．停止時間を短くするためにはバッファ内のオブジェクトを処理するタイミングや一度に処理する量などを制御する必要があり，そのスケジューリングによってスループットは大きく変化する．

本研究では 1 つ目のアプローチをとり，そのスループットを向上させることに焦点をおく．

2.2 Incremental Mark Sweep

Mark sweep（MS）は停止型トレース GC の 1 つである．ヒープが満杯になるとユーザプログラムを止めて，ルートからたどれるオブジェクトを再帰的にマークし（マークフェーズ），マークが終わったらヒープ全体をスイープしてマークされていないオブジェクトを解放する（スイープフェーズ）．MS は循環ごみも解放でき，多くの場合 RC よりもコストが小さいが，停止時間が長い．

Incremental mark sweep（IMS）は MS を変更し，ユーザプログラムの処理とマークを交互に行うことで停止時間を短くしたものである．以下では代表的な IMS アルゴリズムを述べる．ヒープ占有率（=ヒープサイズのうちすでに割り当てられた割合）がしきい値を超えたらマークフェーズをに入る．そしてユーザプログラムによるメモリ割当て要求時に，マーク処理を少しずつ進める．ここで，メモリ不足になる前にマークが終了するように，マーク処理を適切にスケジューリングする必要がある．最悪の場合には全オブジェクトが生存し，それらをすべてマークする必要があるので，ヒープ占有率が 1 になる前にヒープ全体をマークできるようなペースでマーク処理を行う．ただし，多くの場合はそれより早くマークフェーズは終了する．

メモリ不足を防ぐために，典型的には以下のようにマーク量の制御を行う．まずある定数 k を決め，ヒープ占有率が $(k-1)/k$ を上回ったらマークフェーズに入る．マークフェーズではユーザプログラムにより一定サイズのメモリ割当てが行われるとその k 倍のオブジェクトをマークする．

他に，GC 処理を専用スレッドに行わせ，スケジューリングを OS に任せる手法も利用されている．しかしこの場合，メモリ不足や早すぎる GC 終了を防ぐのが困難である．

2.3 Generational GC

Generational GC は我々とは異なるアプローチでスループットを向上させる方式である．一般にプログラム中では長く解放されていないオブジェクトは割り当てられてから間もないオブジェクトよりも後に解放されることが多い，とされている．Generational GC はこの性質を利用したアルゴリズムである．最も単純な Generational GC はヒープを新しいオブジェクトの領域（minor heap）と古いオブジェクトの領域（major heap）の 2 つに分け，heap が不足したら minor heap だけ GC を行い，十分メモリを解放できなかったときに major heap の GC を行う．一般には Generational GC は non-Generational GC よりもスループットが高く，また minor GC は小さい領域の GC なので多くの停止では停止時間が短い．しかし major GC ではヒープ全体の GC を行うことになり，major GC に停止 GC を用いると停止時間が長くなってしまう．

2.4 Ulterior Reference Counting

最後に本研究と同様に短い停止時間で高スループットを達成することを目的とする Ulterior Reference Counting（URC）[10] を紹介する．URC は Generational GC を用い，minor heap は Copying を，major heap は RC を用いる．そして cycle collection には 2.1.2 項で述べた cycle detection をもとにした，Bacon らによる Concurrent Cycle Detection [1] を用いる．URC は Generational GC と RC を用いることで高いスループットと短い停止時間を実現している．しかし，循環ごみの候補の処理に関しては Bacon らは述べておらず，URC では空きメモリが少なくなるとその度合いによってバッファ内のオブジェクトの 12.5%，25%，50%，100%を一度に処理するとしている．そのため循環参照の多いアプリケーションでは最大停止時間が長くなってしまうと考えられる．

3. コストモデル

本章では各 GC 方式のコストを議論するためのモデルを述べる．コストを議論するために，ユーザプログラムの進行をメモリ割当て量で表し，それに対するヒープ占有量とマーク量の遷移を考慮する必要があるが，それが容易に可能なモデルである．

まず，ヒープ使用率はユーザプログラムが各瞬間において使用しているメモリの割合を指す．本モデルでは単純のために，ヒープ使用率とヒープサイズはプログラムの実行を通して一定とする．また，IMS においてはマーク終了後にただちにスイープを終わらせることを仮定する．RC 処理の議論においては使用メモリのうちの循環参照構造の割合はつねに一定であると仮定する．そして循環でないオブジェクトはごみになればただちに解放されるとする．

ヒープサイズを M，プログラム実行全体で生成されるオブジェクトの個数を N とする．また，循環参照構造の割合を c，ヒープ使用率を r_u，ヒープ占有率を r_a とする．ここでヒープ占有率は，ヒープ使用

率に加え，ごみであるがまだ GC によって解放されていない部分を含むものであり，時刻によって遷移する．なお，マークフェーズ終了時のヒープ占有率を特に r_{ae} とする．ここではスイープフェーズでごみを解放してから次のスイープがごみを解放するまでを MS サイクルと呼ぶ．そしてプログラム実行全体で起こる MS サイクル数を n とする．

N，r_u，c はユーザプログラムの性質により決まる値であり，GC アルゴリズムに依存しない．ヒープサイズ M はユーザが与える値であり，これも GC アルゴリズムによらない．一方 r_{ae}．n は GC アルゴリズムによって異なりうる．

3.1 IMS のモデル化

図 1 は IMS におけるヒープ占有量とマーク量の遷移のグラフである．横軸はメモリ割当て量を表し，時刻に対応すると考えてよい．縦軸はメモリサイズを表す．以下，IMS は 2.2 節で述べたように動作するとする．一度の MS サイクルでヒープ占有量とマーク量は以下のように遷移する．まず t_0 でサイクルが開始する．サイクルが開始してからしばらくはマークを開始せず，ヒープ占有量だけが傾き 1 で増加する．ヒープ占有量がしきい値 $M(k-1)/k$ を超えたらマークフェーズに入る（t_1）．マークフェーズ中はマーク量も傾き k で増加する．マーク量がヒープ使用量に達する（t_2）とマークフェーズが終了し，スイープを行う．この時点でごみが解放されるので，ヒープ占有量は減り，ヒープ使用量と同じになる．

グラフにおいては，マークフェーズが終了する時点でヒープは満杯になっていない．この理由は，GC はヒープ使用量を知らないという前提で動作する必要があるからである．さらに GC の処理はヒープ使用量 Mr_u が M に等しいような最悪の場合でもメモリ不足にならないようにしなければならない．グラフ中の破線は仮にヒープ使用量がより高かったときの占有量とマーク量を示すが，この 2 線はちょうど M において交差している．つまり，上述したスケジューリング手法であれば最悪の場合でもメモリ不足にならず，マークがちょうど間に合うといえる☆．通常はヒープ使用量は M より小さいので，ヒープが満杯になる以前（この場合は t_2）にマークは終了する．

また図 1 からヒープ使用率が大きくなると，すなわち r_u が大きくなると $t_1 - t_0$ が小さくなり，1 サイクルの長さが短くなることが分かる．すると，プログラム実行全体でのサイクル数が多くなる．一方，1 サイクルのコストは今の仮定の下ではつねに一定である．以上からヒープ使用率が大きくなるとプログラム実行全体での GC コストが大きくなることが分かる．

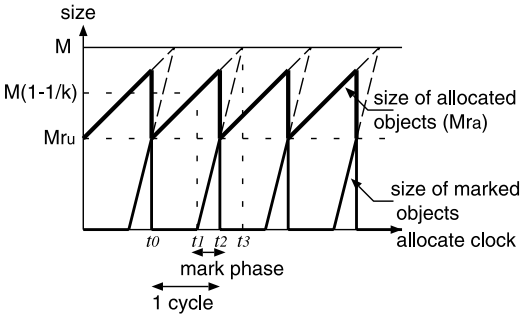

図 1 IMS のヒープ占有量とマーク量
Fig. 1 Size of allocated objects and size of marked ones with simple IMS.

4. 安定したスループットを得る GC アルゴリズム

4.1 提案手法の概要

本研究ではインクリメンタル GC のヒープ使用率の増加にともなうスループット低下を抑える手法を提案する．IMS はヒープ使用率の増加にともない，スループットが極端に低下する．一方，RC では cycle collection を行わないかぎりではスループットはヒープ使用率によらない．しかし RC で cycle collection を行うと 2.1.2 項で述べたように，やはりヒープ使用率の増加にともない cycle collection の処理量が増加するため，その処理を適切にスケジューリングする必要がある．

そこで本研究では，RC を基本とし，バックアップ GC として IMS を用いるハイブリッド GC を採用する．しかし RC と IMS を単純に組み合わせただけでは，特にヒープ使用率の高いときに必要以上に MS サイクルが増加してしまい，スループットが大きく低下することが分かった．この問題を解決するため，IMS のマーク量を動的に制御する手法を提案する．この手法の狙いは，RC が非循環ごみを解放するときは IMS の仕事量を抑制することである．これにより極力 IMS サイクルを減らしてヒープ使用率が高いときのスループット低下を抑えることができる．

4.2 マーク量の動的制御の必要性

前述のように IMS ではユーザプログラムのメモリ割当てとマーク量の比を一定（k）とした．このような静的制御を行う IMS を無変更で RC のバックアップとして用いたハイブリッド GC を，以下では S-Hybrid

☆ ただしこの議論ではフラグメンテーションは起こらないと仮定している

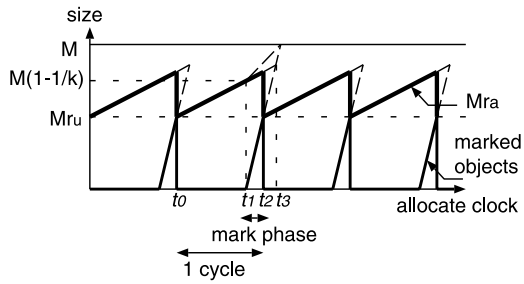

図 2 S-Hybrid GC でのヒープ占有量とマーク量
Fig. 2 Size of allocated objects and size of marked ones with Hybrid GC controlled statically.

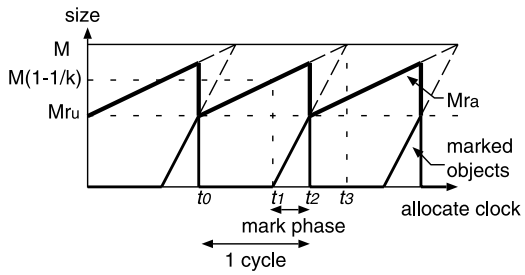

図 3 D-Hybrid GC でのヒープ占有量とマーク量
Fig. 3 Size of allocated objects and size of marked ones with Hybrid GC controlled dynamically.

GC と呼ぶ．S-Hybrid GC では必要以上に MS サイクル数が増加し，スループットが低下してしまうことを以下に示す．

S-Hybrid GC の挙動は**図 2** のようになる．グラフに明示的には表れないが，RC は頻繁に非循環ごみの解放を行っているとする．その解放はメモリ割当て量に対して一定の割合 $(1-c)$ だけ生じ，その結果としてグラフ中のヒープ占有量は傾き c で増えている☆．そしてヒープ占有量が $M(k-1)/k$ を上回ったらマークフェーズに入る．マークフェーズ中のマーク量はつねに傾き k で増える．ここでも図 1 と同様にヒープ使用量が仮に高かった場合の遷移を破線で示す．このとき破線は M より小さい値で交差することが分かる．つまりこの方式はメモリ不足を起こさないが，必要以上にマークが早く終わり，ヒープを十分利用できていない．すると一度の MS サイクルをそれほど長くすることができず，プログラム実行全体でのサイクル数が必要以上に多くなるため，全体のコストが大きくなる．

この問題は，RC による解放がヒープ占有量の増加率を下げているにもかかわらず，IMS がそれを考慮していないことに起因する．そこで，次節ではヒープの空きサイズとマーク量をもとにマーク量を動的に制御することを考える．

4.3 マーク量の動的制御の手法

本手法では IMS の最中に RC がメモリを解放し，ヒープの空きが増加しうることに注目する．ヒープに十分な空きがあるときはマーク処理を遅れさせ，その結果として一度の MS サイクルを長くして全体のコストを小さくすることを目的とする．本節ではコストの削減とメモリ不足の防止を両立させるような制御方法を提案する．この手法を用いるハイブリッド GC を D-Hybrid GC と呼ぶ．

☆ 一般のプログラムでは，循環参照構造の割合 c が一定という仮定は成り立たない．そのため，占有量の遷移は本グラフのように傾きが一定にはならず，増減もしうる．

M_f をヒープの空きサイズ（$= M - Mr_a$）とし，M_r を今後マークしうるオブジェクトの合計サイズの最大値とする．マークフェーズ中に生存オブジェクトの量を見積もることはできないため，M_r はヒープサイズとこれまでマークした量の差とする．また，k を 2.2 節で説明したようにマーク量の制御に用いる定数とする．

マーク量の制御は以下のように行われる．アロケーションの際にその時点の M_f と M_r を調べる．ここで M_f は RC によって解放された量も含んでいる．そして $M_r \leq kM_f$ が成り立つかどうかを調べ，成り立たなければこの式が成り立つようななるべく小さい量だけマーク処理を進める．マーク処理を進めると M_r は減少するため，この不等式はいずれ成り立つようになる．また，この手法によって一度に行われうるマーク量は IMS のものと同等かそれ以下であり，停止時間が短いという要件も満たしている．

この D-Hybrid GC の挙動は**図 3** のようになる．S-Hybrid GC と同様に RC による解放はメモリ割当て量に対してつねに一定の割合 $(1-c)$ だけ生じ，ヒープ占有量は傾き c で増えるものとする．ヒープ占有量が $M(k-1)/k$ を上回ったらマークフェーズに入る．マークフェーズ中のマーク量は先ほど述べた方式により制御されると，傾き ck で増える．これは RC がメモリを解放しているときは一度のマーク量を少なくできることを意味する．さらに，このとき破線は M で交差し，ヒープを過不足なく利用していることが分かる．つまり，メモリ不足と早すぎる IMS 終了の両方を防ぐことができる．この手法により，S-Hybrid GC よりもサイクル長が長くなり，サイクル数を減らすことができるため，コストを低くすることができる．

4.4 停止時間の制御

GC 処理の大きな停止は以下の処理によって生じる．

(1) RC によるデクリメントと解放
(2) IMS によるマーク

(3) IMS によるスイープ

(1) に関しては，デクリメントバッファ（2.1.1 項参照）に一定量のオブジェクトがたまったらルートをスキャンし，一定量だけ参照デクリメントと解放の処理を行うことにより，最大停止時間を保証する．

(2) に関しては，IMS ではオブジェクトをマークするとバッファにため，そのあとバッファからとりだしてその要素のオブジェクトをマークし，バッファにためる，というように再帰的に処理する．そこで，4.3 節で述べたように占有率とマーク量からマーク処理を行うタイミングを求め，一度の処理で一定時間マーク処理を行うことにより最大停止時間を保証する．

(3) に関しては，lazy sweep を用いることで最大停止時間を保証する．ただし，現在は未実装のまま実験を行っている．

4.5 バッファ内のオブジェクトの同期問題

RC はデクリメントされるオブジェクトをバッファにためる．IMS はマークしたオブジェクトを別のバッファにため，その後バッファからとりだしてその要素をマークする．このように 2 つの GC が別々のバッファを用いて独立に解放を行う場合，一方の GC のバッファ内のオブジェクトをもう一方の GC が解放してしまうと，バッファからそのオブジェクトのアドレスをとりだしてもそのアドレスにはすでに求めるオブジェクトが存在しないことがある．この問題を解決するために，バッファ内のオブジェクトの同期をとる必要がある．具体的には，マークされたオブジェクトを RC で解放しないこと，また IMS のスイープの前に RC のデクリメントの処理を行うこととした．

5. モデルに基づくコストの見積り

3，4 章で提案したモデルを用いて IMS，S-Hybrid GC，D-Hybrid GC のコストの見積りを求める．GC のコストは IMS のマークコスト（W_m），スイープコスト（W_s），RC の解放コスト（W_r）から成り立つとし，それぞれユーザプログラムの実行全体における合計を考える．以下にそれぞれのコストの算出法を述べる．

- マーク：マークのコストは生存オブジェクト数とサイクル数に比例する．そこで 1 つのオブジェクトをマークするコストを m とすると，$W_m = Mr_umn$ となる．
- スイープ：スイープのコストはマークフェーズ終了時の占有オブジェクト数とサイクル数に比例する．そこで 1 つのオブジェクトをスイープするコストを s とすると，$W_m = Mr_{ae}sn$ となる．
- RC のコスト：RC の処理は参照回数の維持とオブジェクトの解放に分けられる．特に参照回数の維持のコストはアプリケーションによって大きく異なるため正確に見積もることはできない．ここでは単純化のために両者の合計が比例係数 r で解放するオブジェクト数に比例するとし，$W_r = N(1-c)r$ とする．

これらをもとにして，それぞれのアルゴリズムにおける合計 GC コストを求めることが可能である．

5.1 IMS

図 1 より，IMS では 1 サイクルの長さは $M(r_{ae}-r_u)$ となり，サイクル数 n は

$$n = \frac{N}{M(r_{ae} - r_u)} \tag{1}$$

となる．r_{ae} はマークフェーズ開始時は $(k-1)/k$，マークフェーズ中の増加分は r_u/k であるので

$$r_{ae} = \frac{k-1}{k} + \frac{r_u}{k} \tag{2}$$

よって，IMS のコストは

$$W_m = \frac{Nkr_um}{(k-1)(1-r_u)} \tag{3}$$

$$W_s = \frac{N(k-1+r_u)s}{(k-1)(1-r_u)} \tag{4}$$

$$W = \frac{N\{kr_um + (k-1+r_u)s\}}{(k-1)(1-r_u)} \tag{5}$$

5.2 S-Hybrid GC

図 2 より 1 サイクルの長さは $M(r_{ae} - r_u)/c$ となり，サイクル数 n は

$$n = \frac{Nc}{M(r_{ae} - r_u)} \tag{6}$$

となる．r_{ae} はマークフェーズ開始時は $(k-1)/k$，マークフェーズ中の増加分は cr_u/k であるので

$$r_{ae} = \frac{k-1}{k} + \frac{cr_u}{k} \tag{7}$$

よって，S-Hybrid GC のコストは

$$W_m = \frac{Nckr_um}{k-1+cr_u-kr_u} \tag{8}$$

$$W_s = \frac{Nc(k-1+cr_u)s}{(k-1+cr_u-kr_u)} \tag{9}$$

$$W_r = N(1-c)r \tag{10}$$

$$W = \frac{Nc\{kr_um + (k-1+cr_u)s\}}{(k-1+cr_u-kr_u)} + N(1-c)r \tag{11}$$

5.3 D-Hybrid GC

図 3 より 1 サイクルの長さは $M(r_{ae} - r_u)/c$ となり，サイクル数 n は

$$n = \frac{Nc}{M(r_{ae} - r_u)} \tag{12}$$

となる．r_{ae} はマークフェーズ開始時は $(k-1)/k$，マークフェーズ中の増加分は r_u/k であるので

$$r_{ae} = \frac{k-1}{k} + \frac{r_u}{k} \tag{13}$$

よって，D-Hybrid GC のコストは

$$W_m = \frac{Nckr_um}{(k-1)(1-r_u)} \tag{14}$$

$$W_s = \frac{Nc(k-1+r_u)s}{(k-1)(1-r_u)} \tag{15}$$

$$W_r = N(1-c)r \tag{16}$$

$$W = \frac{Nc\{kr_um + (k-1+r_u)s\}}{(k-1)(1-r_u)} + N(1-c)r \tag{17}$$

5.4 性能予測

以上の結果から導出した 3 種類の GC のコストの理論値を**図 4** にまとめた．この図は縦軸にコストをとり，横軸にヒープサイズと必要な最小ヒープサイズとの比（以下，相対ヒープサイズ）をとる．マーク，スイープ，RC 処理のコストとしてここでは $m : s : r = 3 : 1 : 8$ とした．また IMS スケジューリングの定数 $k = 4$，循環参照構造の割合 $c = 0.3$ とした．グラフから分かるように，どの手法でも相対ヒープサイズが小さいほど GC コストが増加するという傾向が見られる．その中で，D-Hybrid GC はコスト増加を最も抑えることができることが分かる．上述の条件においては，相対ヒープサイズが約 1.7 以下の場合に，単独の IMS よりコストが低い．また，この図から D-Hybrid GC の曲線は S-Hybrid のものに比べて左に平行移動したものとなっている．この点から，動的制御を行うことにより同一のスループットを得るために必要なメモリが約 20%少なくて済むととらえることができる．

6. 実験と評価

本章では D-Hybrid と S-Hybrid，IMS，cycle collection を行わない単独の RC の性能比較を行う．また，停止 GC である MS，Generational Copy-MS（GenMS）の性能も示す．D-Hybrid と S-Hybrid を IBM で開発されたオープンソースの Java VM である Jikes RVM [3] のメモリ管理モジュールである JMTk [9] に実装した．もとの JMTk には IMS が実装されていなかったため，IMS の実装も行った．その他の GC は JMTk に実装されているものを用いた．ベンチマークとして SPECjvm98 ベンチマーク [7] の_213_javac と_228_jack，DaCapo プロジェクトの DaCapo ベンチマーク [8] の antlr，jython，ps を用いた．**表 1** にそれぞれのベンチマークの Jikes RVM 上での使用メモリサイズ（GenMS でメモリ不足になる最大のヒープサイズ）と循環参照の割合を示す．循環参照の割合は以下のように求めた．実験により，MS が一度も GC を行わない最小のヒープサイズ（M_{max}）を求めた．この値は生存オブジェクト，循環ごみと非循環ごみのサイズの和である．次に RC がメモリ不足になるような最大のヒープサイズ（M_{cycle}）を求めた．これは生存オブジェクトと循環ごみのサイズの和である．最後に，MS がメモリ不足になる最大のヒープサイズ（M_{min}）はおよそ生存オブジェクトのサイズに等しい．以上から，$(M_{cycle} - M_{min})/(M_{max} - M_{min})$ を循環参照の割合とした．

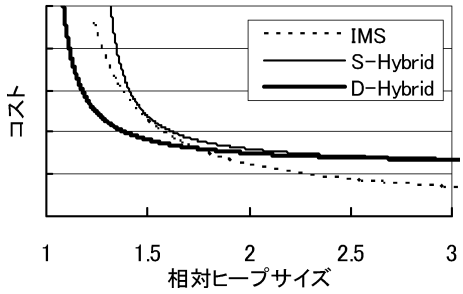

図 4 コストの予測
Fig. 4 Calculation of costs.

表 1 各ベンチマークの使用メモリサイズと循環参照の割合
Table 1 Memory size and cycle rate of benchmarks.

benchmark	メモリサイズ [MB]	循環参照の割合
_213_javac	27	0.35
_228_jack	9	∼0.02
antlr	13	0.2
jython	13	∼0.02
ps	21	∼0.02

表 2 _228_jack の停止時間
Table 2 Pause time of _213_javac.

GC	平均 [ms]	最大 [ms]	停止回数
MS	1,089	1,102	4
GenMS	32.2	1,135	379
IMS	17.0	32	2,490
RC	30.1	50	627
S-Hybrid	30.2	115	1,345
D-Hybrid	30.5	110	979

6.1 停止時間

停止時間の平均値，最大値はメモリサイズによって異なるが，ほぼ同じような傾向が見られた．**表 2** に_213_javac のヒープサイズが 52 MB のときの停止時間を示す．MS は停止時間が長く，GenMS では ma-

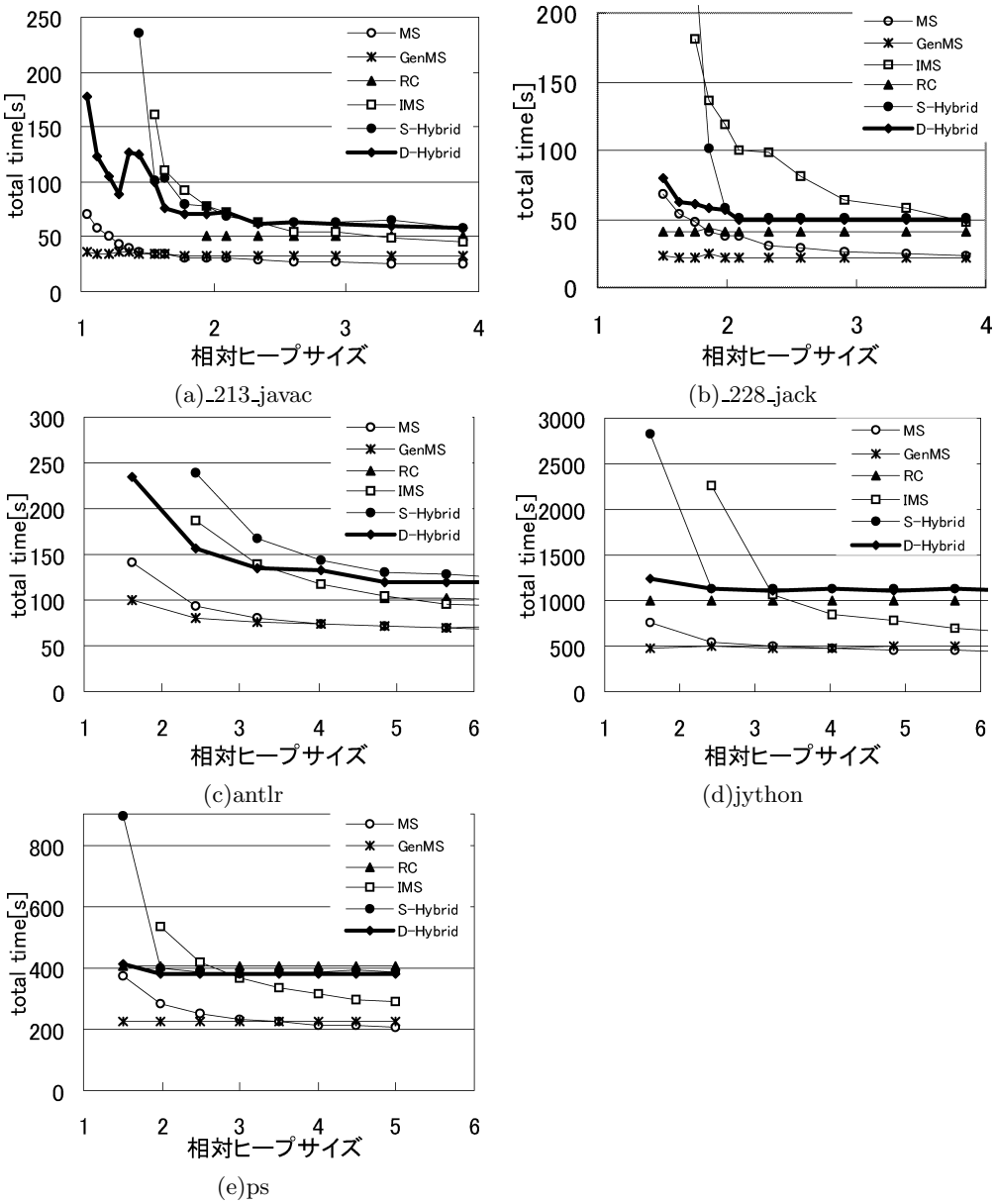

図 5 相対ヒープサイズと総実行時間の関係．横軸はアプリケーションが使用する最小メモリサイズに対するヒープサイズの割合

Fig. 5 Relationtotal between relative heap size and total time. The horizontal axis indicates relative heap size to minimum heap size used by an application.

jor GC での停止時間が長い．一方，IMS や RC は最大停止時間が短い．S-Hybrid，D-HybridGC は平均時間は短いが，最大停止時間は IMS や RC ほど短くはない．この理由は現在の実装が以下の 2 点において不十分なためであり，将来の改良により改善可能である．まず 1 点目は現在は lazy sweep を実装しておらずスイープ時の停止時間が長いことがあげられる．lazy sweep を実装すれば，スイープのための停止時間を 30 ms 以下に抑えられることは確認できている．もう 1 点は RC の再帰的な解放による停止があげられる．RC の解放は 4.4 節で述べたように，一定時間だけデクリメントをするが，4.5 節で述べたようにスイープ前だけはデクリメントバッファ内のオブジェクトをすべてデクリメントする必要があるため，再帰的な解放を一度に処理している．この再帰的な解放によって長い停止が生じてしまう．再帰的な解放はデータ構造に

よっては非常に長い停止につながってしまうため，今後解決すべき課題である．

6.2 総コスト

IMS や RC の GC 処理は実行プロセス中に細かく分散して行われるため，GC 処理とユーザプログラムの処理を明確に分けて評価することが困難である．そのため本節では，アプリケーションの総実行時間を比較することによって GC コストを評価する．各ベンチマークでの総実行時間を**図 5** に示す．図 5 は縦軸に総実行時間を，横軸に図 4 と同様にヒープサイズと必要な最小ヒープサイズとの比（相対ヒープサイズ）をとった．横軸の値が小さいほどヒープが小さく，ヒープ使用率が高くなるため，総実行時間が増加する傾向を持つ．各ベンチマークでヒープサイズが最大のときは，MS では一度もマークフェーズに入っていないので，このときの MS の実行時間はアプリケーションの処理時間に近いものになっている．

まず RC と D-Hybrid の比較を行う．D-Hybrid は RC の処理に加えて IMS の処理を行うため，RC よりコストが小さくなることはない．しかし循環参照の多い_213_javac と antlr ではヒープサイズが小さいときに RC の実行はメモリ不足のため失敗してしまうが，D-Hybrid は IMS と同等かそれよりヒープサイズが小さいときでも実行可能であった．

次に，IMS と S-Hybrid，D-Hybrid の比較を行う．まず，総じて IMS，S-Hybrid ではヒープサイズが小さくなるにつれて極端に実行時間が増加することが分かる．それに比べて，D-Hybrid では実行時間の増加が抑えられている．アプリケーションごとに詳しく調べると，循環参照が少ない_228_jack，antlr，ps については相対ヒープサイズが 3 以下のとき IMS より D-Hybrid のコストが小さくなった．S-Hybrid は相対ヒープサイズが 2 以上では D-Hybrid と似た性能だが，それより小さくなると IMS と同様に実行時間の増加が著しい．一方，循環参照が比較的多い_213_javac と jython については相対ヒープサイズが 2 以下で IMS より D-Hybrid のコストが小さくなった．S-Hybrid も IMS と同程度かそれ以上に実行時間が増加する．循環参照の割合が多いアプリケーションでは D-Hybrid でのマーク処理量が多くなり，実行時間の変動が大きくなるが，それでも IMS よりは抑えられることが分かる．以上の結果は 5 章での予想とも一致しており，本研究の目的を達成しているといえる．

最後に Generational GC に関して考察を行う．GenMS はほとんどの場合に最もスループットが高く，antlr 以外ではヒープサイズに対して安定であり，GC のコストは非常に小さい．しかし，今回用いた GenMS は major GC による停止時間が長い．これをインクリメンタルにした場合は MS と IMS の比較から類推されるように，ヒープ使用率の上昇にともなうスループットの低下が生じると考えられる．そのような場合に major GC として我々の D-Hybrid を用いることにより，スループットの低下を軽減できると考えている．

7. おわりに

本稿ではヒープ使用率の上昇にともなうスループットの低下を抑える GC を提案した．その基本は RC と IMS を組み合わせたハイブリッド GC であり，IMS のマーク量を動的に制御することによりスループット低下を防ぐものである．そして IMS，静的制御のハイブリッド GC（S-Hybrid），動的制御のハイブリッド GC（D-Hybrid）に関してモデルを用いてコストを算出，比較した．その結果，D-Hybrid GC ではより安定したスループットを得られ，同一スループットを実現するのに必要なヒープサイズを削減する効果を持つことを理論的に示した．さらに提案手法を Jikes RVM に実装して，実験結果から本手法が短い停止時間と安定したスループットを実現することを示した．今回実験した 5 個のベンチマークすべてにおいてヒープ使用率が高いときに IMS より高速であることが確認された．

今後の課題として停止時間の向上に関しては lazy sweep の実装，バッファの同期と RC による再帰的解放への対処があげられる．スループットの向上に関してはまず本手法と generational GC を組み合わせることがあげられる．また，本研究の手法はヒープ使用率が低いときは IMS よりコストが高くなることが多い．これを改善するため，ヒープ使用率が高いときだけ RC 処理を行う手法を考案したい．具体的には，IMS のマーク時に参照回数を数えてマーク終了時に参照回数を補完し，スイープ後にヒープ使用率が高ければ RC を再開することが考えられる．これによりヒープ使用率が低いときは IMS と同程度のコストに抑えられると考えられる．

参考文献

1) Bacon, D.F. and Rajan, V.T.: Concurrent Cycle Collection in Reference Counted Systems, *Proc. 15th European Conference on Object-Oriented Programming* pp.207–235 (*ECOOP '01*), Springer-Verlag, London, UK (2001).
2) DeTreville, J.: Experience with garbage collection for modula-2+ in the topaz environment,

OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems (1990).
3) Jikes $^{TM}Research Virtual Machine(RVM)$. http://jikesrvm.sourceforge.net/
4) Jones, R. and Lins, R.: *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*, Wiley & Sons (1996).
5) Deutsch, L.P. and Bobrow, D.G.: An Efficient Incremental Automatic Garbage Collector, *Comm. ACM*, Vol.19, No.9, pp.522–526 (1976).
6) Lins, R.D.: Cyclic Reference Counting with Local Mark-Scan, *Inf. Process. Lett.*, Vol.44, No.4, pp.215–220 (1992).
7) Standard Performance Evaluation Corporation: SPECjvm98 benchmarks. http://www.spec.org/osg/jvm98/
8) McKinley, K.S. and Blackburn, S.M.: The DaCapo benchmarks. http://www-ali.cs.umass.edu/DaCapo/
9) Blackburn, S.M., Cheng, P. and McKinley, K.S.: A Garbage Collection Design and Bakeoff in JMTk: An Efficient Extensible Java Memory Management Toolkit, TR-CS 03-02, ANU (Sep. 2003).
10) Blackburn, S.M. and McKinley, K.S.: Ulterior Reference Counting: Fast Garbage Collection without a Long Wait, *OOPSLA '03: Proc. 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages and applications*, pp.344–358, ACM Press, New York, NY, USA (2003).

(平成 17 年 7 月 5 日受付)
(平成 17 年 11 月 9 日採録)

白井 達也（学生会員）
1983 年生．2005 年東京大学工学部電子情報工学科卒業．同年より東京大学大学院情報理工学系研究科電子情報学専攻修士課程在学中．高速計算の基盤技術に興味を持つ．ACM 学生会員．

遠藤 敏夫（正会員）
1974 年生．2001 年東京大学大学院理学系研究科情報科学専攻博士課程修了．博士（理学）．日本学術振興会特別研究員，科学技術振興機構研究員等を経て，2004 年より東京大学大学院情報理工学系研究科特任助手．主に分散/並列処理，言語処理系の研究に従事．日本ソフトウェア科学会，ACM，IEEE-CS 各会員．

田浦健次朗（正会員）
1969 年生．1997 年東京大学大学院理学博士（情報科学専攻）．1996 年より東京大学大学院理学系研究科情報科学専攻助手．2001 年より東京大学大学院情報理工学系研究科電子情報学専攻講師．2002 年より同助教授．日本ソフトウェア科学会，ACM，IEEE-CS，USENIX 各会員．

近山 隆（正会員）
1953 年生．1977 年東京大学工学部計数工学科卒業．1982 年東京大学大学院情報工学専門課程修了．工学博士．同年より ICOT において第五世代コンピュータプロジェクトに参加．1995 年より東京大学に移り，現在同新領域創成科学研究科基盤情報学専攻教授．日本ソフトウェア科学会，人工知能学会，ACM 各会員．