プログラミング 16-5
(1997. 11. 21)

実時間ゴミ集めにおけるルート挿入の効率化

近藤 豪† 中西正和††

†慶應義塾大学大学院 理工学研究科 計算機科学専攻
††慶應義塾大学大学院 理工学部 情報工学科

ゴミ集め (以下 GC) によって生じる処理の停止時間をなるべく抑えるための研究として，実時間 GC がある．この実時間 GC の主要なアルゴリズムである Snapshot-at-beginning では，GC 開始時に行なうルート挿入の際に一旦処理を停止させなければならない．現在この停止時間は，ルート集合の大きさに依存し，上限が定まっていない．これは実時間システムにとっては致命的な欠点であると考えられる．

本稿では，この停止時間に対して上限を定める，インクリメンタルルート挿入と呼ばれる手法を提案する．そして，その手法を Lisp1.5 をベースとする言語処理系のインクリメンタル GC 上に実装し，その有効性を実験によって示した．

Efficiency Improvement of Root Insertion on Real-time Garbage Collectors

Go KONDO† Masakazu NAKANISHI††

†Department of Computer Science
Graduate School of Science and Technology
†† Department of Information and Computer Science
Faculty of Science and Tchnology
Keio University
3-14-1, Hiyosi, Kouhoku, Yokohama 223, Japan

Real-time Garbage Collection is a study that makes pause time caused by GC as short as possible. But, Snapshot-at-beginning algorithm, which is a major algorithm of real-time GC, has to pause processing while root insertion at the beginning of GC. That pause time depends on the scale of root set, so its upper bound is not fixed. It may be a fatal weak point for real-time system.

In this paper, we propose a method called "Incremental Root Insertion", that fixes upper bound of the pause time. Then, we implemented the method on a incremental garbage collector of the Lisp1.5 based system, and showed how efficient that is by a few experiments.

1 はじめに

Lispやオブジェクト指向プログラミング言語などの処理系では，動的に確保されたメモリ領域を再利用するゴミ集め(以下GC)が必要となる．通常はGCを起動する際に処理を停止し，GCが終了するまで待つということをする．しかし，この中断時間は実時間性が要求されるアプリケーションの実行では致命的な欠点となる．この中断時間をなるべく短くすることを目的とした研究が実時間GCである．

実時間GCではこれまで様々な手法が提案されてきたが，その主要なアルゴリズムであるSnapshot-at-beginning[5]では，ルート挿入時の停止時間に対して上限が定まっていない．本稿では，まずはじめにに実時間GCのアルゴリズムとその問題点をのべ，次にその問題点を解消するインクリメンタルルート挿入(以下**IRI**)と呼ばれる手法の提案を行なう．そして最後に，その手法を実装したインクリメンタルGCでの実験結果を報告し，その有効性を検証する．

2 実時間GC

実時間GCでは，以下のようなことが目的となる．

1. メモリ領域の枯渇による処理の停止を防ぐ
2. 毎回のGCを実行する際になるべく処理を中断させない．

この目的を達成するために，GCプロセス**collector**とリスト処理プロセス**mutator**を並列に動作させる**並列GC**とmutatorの処理中にcollectorの処理を時間的に分散させ，疑似的にmutatorが停止していない様に見せる**インクリメンタルGC**の2通りの方法で実現される．両者は採用するアルゴリズムや実装上の問題点などは共通しているので，本稿では区別せず，実時間GCとして議論する．

2.1 実時間GCのアルゴリズム

まず，問題点を明らかにするために，collectorによる生きているオブジェクトの走査を3色の印づけで説明する．**black**はすでに印づけを行なったオブジェクト，**gray**は印づけを行なったが，その先はまだ走査していないオブジェクト，**white**は印づけがされていないオブジェクトである．一般には，blackが直接whiteを指すことはない．

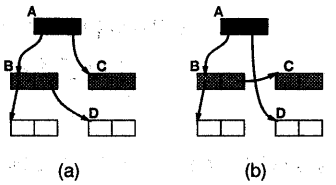

図1: mutatorによるポインタの書き換え

ここで，mutatorがポインタの書換えを行なった場合を考える．mutatorは図1(a)のようなポインタを図1(b)のように変化させた．このような場合，collectorはオブジェクトDが生きていることを認識するためにmutatorとcollectorを協調させなければならない．その方法に2通りあり，一方は，書き込みの際に同期操作を行なうライトバリア型のアルゴリズム，もう一方は，読む際に同期操作を行なうリードバリア型のアルゴリズムである．読み込みの頻度は書き込みの頻度よりも非常に多いためリードバリア型は実行の効率が非常に悪い．そのため，ライトバリアを用いた方法が主流となっており，

- Incremental update
- Snapshot-at-beginning

の2種類がある．

On-the-fly GC[1]によって提案された前者のアルゴリズムは，ポインタの書換えが起こったときには，書き換えた先のオブジェクトを生きているものとして扱う．図1の例では，Aにあるを指しているポインタを切断して，Dを指したときにDをgrayにする．これによって，Dが生きていることが保証される．

しかし，Incremental-updateアルゴリズムでは，ルート挿入の際にmutatorを止める必要がないなどの利点がある一方，頻繁に書き変わるルートセットを印づけが終了する度に，whiteを指していないかチェックする必要がある．これを**root set scanning**

と呼び，Incremental updateアルゴリズムの大きな欠点となっている[2][4].

Snapshot GC[5]によって提案されたこのアルゴリズムは，ポインタの書換えが起こったときには，書換えの前に指されていたオブジェクトを生きているものとして扱うというアルゴリズムである．これによって，GCが始まった時点で生きていたすべてのオブジェクトがルートから到達可能となることを保証している．

図1の例では，BにあるDを指しているポインタを切断したときにDをgrayにする．これによって，Dが生きていることが保証される．

2.2 ルート挿入時における問題点

前述のSnapshot-at-beginning型アルゴリズムは，Incremental-updateよりも回収効率が悪くルート挿入時に停止が必要という欠点があるが，root-set-scanningのような致命的な弱点がなく簡単に実装できるので，現在の実時間GCはほとんどがこのアルゴリズムを採用している．

実時間GCでは一般的に停止型のGCよりも回収効率が悪く，その中でもSnapshot-at-beginningは，特に悪い．これまでの研究では，そのSnapshot-at-beginning型の回収効率を何とかIncremental-update並に引き上げる研究が主であった．成果として，世代別の考えを採り入れた**Partial Marking GC**[3]や，Incremental-updateとSnapshot-at-beginningを組み合わせた**Complementary GC**[2]などがあげられる．これらによって，実時間GCの**目的1**は，達成されたといってよい．

しかし，これらのGCではSnapshot-at-beginningを基本としているため，ルート挿入時にmutatorの停止が必要であり，**目的2**が達成されていないのが現状である．このルート挿入時の停止時間は，ルート集合の大きさに依存する．ルート集合には，レジスタ，グローバル変数，スタックなどが含まれるが，その大部分を占めるスタックは実行時に深さが大きく変化するので，この停止時間に対して上限が定まっていないのが現状である．

スタックに割り付けるオブジェクトをなるべくヒープに割り付けてルート集合をなるべく小さくとるという回避方法もあるが[4]，根本的な解決策にはなっていない．

3 インクリメンタルルート挿入 (IRI)

以上のことから，ルート集合の大部分を占めるスタックが問題となって，ルート挿入時の停止時間に上限が定まっていないことがわかる．しかし，スタックの性質に注目するとmutatorの読み込みと書き込みはトップ付近に注目していることが分かる．

そこで，我々はその性質に着目し，インクリメンタルルート挿入(以下IRI)と呼ばれる手法を考案した．従来の実時間GCでは，GCが起動されるとmutatorを停止させ，ルート集合すべてをcollectorのGCスタックにコピーしていた．IRIでは，ルート挿入をmutatorを停止することなく並列GCの場合はcollectorがmutatorに対して並列に，インクリメンタルGCの場合はcollectorがインクリメンタルに行なう．

ただし，ルート挿入はルート集合に対して，ページ単位で書き込み保護をかけ，保護がかかった部分から挿入をする(図2(a))．挿入の順番は，最適化されたIncremental updateアルゴリズム同様，スタックの下部から上部へと行なう．ここで，mutatorが初めて保護がかけられたページに書き込みを行なった場合，mutatorは止まらなければならない(図2(b))．

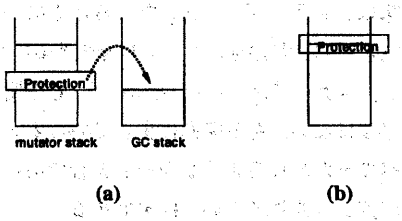

図2: ページ単位のルート挿入

そして，そのページ中にあるルートを挿入する．

また，グローバル変数や，レジスタなどをルートで持っている場合は，それらも挿入する．その後，処理を再開するが，mutatorが停止していた時間の上限は，1ページ分(とレジスタ，グローバル変数)のルート挿入に要する時間であるので，これは定数時間である．

3.1 IRIを採用できる条件

ただし，現在保護をかけてコピーを行なっているページよりも下部のページにmutatorが書き込みを行なった場合に停止時間の上限が保証できるかどうかが問題となる．このケースは，親の局所変数の参照呼びや，ページサイズよりも大きなオブジェクトをスタック上に割り付けた場合などである．これらのケースは，プログラムのスタイルや処理系の実装方法を変更することで回避する．本稿での実験で実装したLispインタプリタは，このようなケースが発生しない処理系である．

3.2 IRIを採用したGCの正当性

GCは，生きているオブジェクトは決してゴミとして回収されない，死んでいるオブジェクトはいつか回収される，という2つのことを満たすとき，正当であるという．

Snapshot-at-beginning型のアルゴリズムは，GCが開始された時点で生きているオブジェクトのスナップショットをとり，それらのオブジェクトは，GCが終了するまでたとえ死んでも生きているとみなす，という考え方に基づいて正当性を保証している．

従来は，スナップショットをとるためにmutatorを停止してGC開始時のルート集合を獲得していた．しかし，スナップショットをとるのは，GC開始時である必要はない．ある瞬間に生きているオブジェクトすべてをたどれるようなルート集合であればGC開始時点より遅れてもいいはずである．

IRIを採用したGCは，書き込み保護に抵触した時点での生きているオブジェクトのスナップショットをとっているという意味で，正当性を満たしたGCである，ということができる．

4 実験結果及び考察

4.1 実験方法

本稿の実験では，Lisp1.5ベースの処理系(セル数1M個)にIRIを採用したインクリメンタルGCとIRIを採用していないインクリメンタルGCを実装し，各種ベンチマークプログラムを用いてルート挿入の時間とその時のスタックの深さおよび全体の実行時間を計測した．IRIでの保護単位になるページのサイズは，OSが定めているページサイズ4096バイトとした．そのときの実験環境および，ベンチマークプログラムは以下の通りである．なお，実行はリアルタイムスケジューリングクラスで行なった．

実験環境

- 機種 — SPARC station 20
- 主記憶 — 96MB
- 2次キャッシュメモリ — 1MB
- OS — SunOS 5.5.1

ベンチマークプログラム

- 再帰による階乗
  再帰での計算のためスタックがはじめに一気に伸び，あとは徐々に縮まる．Bignum計算のためメモリの消費量が多く，GCがよく起動される．
- Bit
  リスト処理プログラム．生きているオブジェクトが多く，消費も大きいためGCがよく起動される．また，スタックは徐々に深くなるという特徴をもつ．
- アッカーマン関数
  数値計算プログラム．スタックの深さ，変動量ともに大きい．

4.2 結果及び考察

4.2.1 ルート挿入時間の比較

IRIを採用したGCと採用しないGCの双方で上記のベンチマークプログラムを実行した際のルート挿入時のmutatorの停止時間とその時のスタックの深さとの関係を図3，図4，図5に示す．なお，インクリメンタルGCでの実験のためIRIではルート挿入時のmutatorの停止が複数回になる．以下の図ではその停止すべてを記録してある．

これらの図を見てわかるとおり，従来型では，スタックが深くなればなるほど停止時間が大きくなっている．しかし，IRIではスタックの深さによらず，停止時間が1ミリ秒以下になっていることがわかる．これらの実験結果からIRIでは，Snapshot-at-beginning型の欠点であるルート挿入時の停止時間に対して上限を保証していることがわかる．

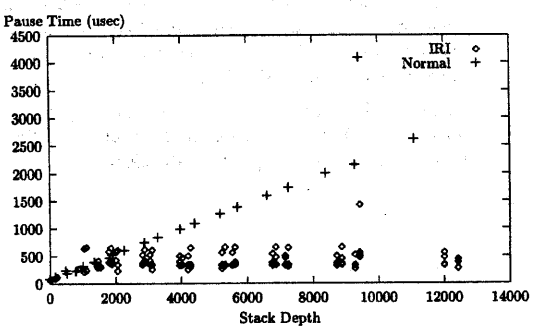

図 3: 再帰による階乗

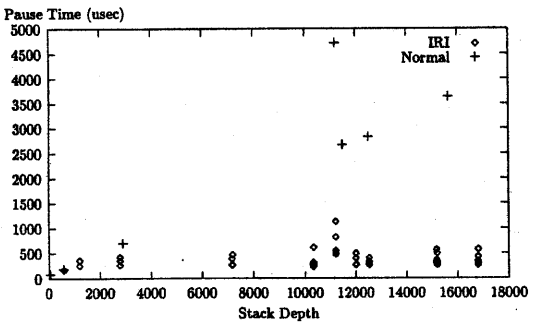

図 4: Bit

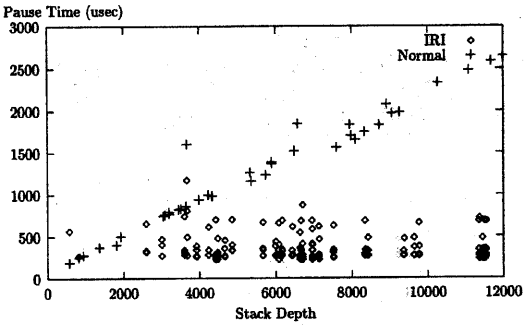

図 5: Ack

これらの実験では，従来型，IRIともに1箇所のみ通常よりも多くの時間がかかってしまう現象が確認された(図3ではスタックの深さが9500付近，図4では11000付近，図5では，4000付近)．その現象が起こるのは決まって1回目のGCのときである．キャッシュミスによる遅延が発生しているものと思われる．

4.2.2 実行時間の比較

ルート挿入時のmutatorの停止時間は上記のとおりであるが，全体の実行時間についても測定を行なった．再帰による階乗，Bit，アッカーマンの結果をそれぞれ，表1，表2，表3に示す．

表 1: 再帰による階乗

	実行(秒)	GC(秒)	起動回数
IRI	32.68	13.00	6
従来型	24.39	9.04	4

表 2: Bit

プログラム	実行(秒)	GC(秒)	起動回数
IRI	17.12	6.13	2
従来型	16.09	5.92	2

この結果から分かる通り，IRIは従来型よりも最大で3割程度遅くなっている．その主な原因がGCの起動回数であると考えられる．IRIではルート挿入フェーズ中にアロケートされるオブジェクトは生きているものとして扱われるので，単純なSnapshot-at-beginningアルゴリズムよりも保守性が大きい．

表 3: アッカーマン

プログラム	実行 (秒)	GC(秒)	起動回数
IRI	238.3	76.13	34
従来型	176.4	44.38	19

よって，そのことが回収効率に影響しているものと思われる．

5 結論及び今後の展望

5.1 結論

以上をまとめると以下が結論として導ける．これよりIRIは有効な手法であると考えられる．

- Snapshot-at-beginning 型アルゴリズムの GC に IRI を実装することで，ルート挿入時の停止時間に対して，上限を保証することが可能となった．このことによって実時間システムへの応用が期待される．
- IRIの導入により，オブジェクトの保守性は高くなる．よって GC の起動回数および実行時間が大きくなる場合がある．

5.2 今後の展望

上より，IRIによってルート挿入時の実時間性を保証することが可能となったが，まだ解決しなければならない問題は多い．それを以下に挙げる．

- 保守性の削減
  IRIでは，オブジェクトの保守性が増し，GC起動回数が増えるという現象が観測された．この保守性を削減することによってIRIの導入コストを抑制することが可能となる．
- 並列 GC への適用
  インクリメンタル GC と並列 GC では，根本的な大ききな違いはない．本稿ではインクリメンタル GC にIRIを実装したが，並列 GC においても実現可能であると考えられる．
- ページサイズの選定
  理論的には，保護をかけるページのサイズが小さければ小さいほど IRI での停止時間は短いはずである．保護をかけるための処理とのトレードオフを考慮に入れた最適なページサイズの選定を調べる必要がある．

参考文献

[1] E. W. Dijkstra, L. Lamport andA. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, Vol. 21, No. 11, pp. 966–975, 1978.

[2] S. Matsui, Y. Tanaka, A. Maeda, and M. Nakanishi. Complementary garbage collector. In *International Workshop on Memory Management*, No. 986 in Lecture Notes in Computer Science, pp. 163–177, 1995.

[3] Y. Tanaka, S. Matsui, A. Maeda, and M. Nakanishi. Partial marking gc. In *Proceedings of International Conference on CONPAR94-VAPP VI*, pp. 337–348, 1994.

[4] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *International Workshop on Memory Management*, No. 637 in Lecture Notes in Computer Science, pp. 1–42, 1992.

[5] T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, Vol. 11, No. 3, pp. 181–198, 1990.