プログラミング 10−6
(1996. 11. 1)

双方向環状リストを用いた世代別並列ゴミ集め処理

荻原 拓也　近藤 豪　田中 詠子　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

Lispをはじめとするリスト処理言語においてゴミ集め処理(以下GC)によって生じる中断時間をなるべくなくすようにするための研究として実時間GCの研究がある．その中でも特に並列GCについては現在まで数多くの手法が提案されてきた．しかし，双方向リストを用いたTreadmillという手法は有望視されながらもリスプ処理系に実装された例は今のところない．そこで本稿ではLisp1.5ベースの処理系にTreadmillを実装し並列化を行なう．さらに世代別GCの考え方を採り入れたGenerational Treadmillという手法を提案，評価も行なった．従来のTreadmillに世代別GCの考え方を採り入れることにより実験を行なった結果，実時間GCにとって重要であるセルの回収率，1回のGCにかかる時間などの面でTreadmillアルゴリズムの有効性を示すことができた．

The Generational Parallel Garbage Collection Using Double Linked Cyclic Lists

Takuya OGIHARA　Go KONDO　Eiko TANAKA
Masakazu NAKANISHI

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

Parallel garbage collection, which executes list processing and garbage collection(GC) in parallel, has a great potential for real-time GC. So, various Parallel GC techniques have been researched by many scholars in the world. In these situation, there is no report of one of promising GC technique "Treadmill" on Lisp System. In my research, the parallel GC using double linked cyclic list (Treadmill GC technique) on lisp system, is considered and explained. Moreover, We propose Generational Treadmill ,appending Generational technique to Treadmill, and this technique is considered and explained.

1 はじめに

Lispなどのシステム上で実時間処理を行なう際に必要な実時間GCは，現在ではリスト処理プロセス(以下mutator)とGC処理プロセス(以下collector)を並列に実行する並列GCという手法が一般的である．collectorはmutatorを停止させることなくフリーセルを安定供給しなくてはならないため回収する際の効率などが問題となる．

並列GCにおいて使用されるアルゴリズムはそのほとんどがマークスイープ法をベースとしている[2, 4]. 逐次処理系では非常に優秀な結果を示すコピー法は，並列化することによってリードバリア[1]を生じてしまうためあまり利用されることはなかった．しかし，Treadmill[3]というコピー法をベースとしたリードバリアが生じることのない並列GCの手法が考案されたにもかかわらず，この手法を用いてLisp処理系に実装したという報告は現在のところはない．

そこで本研究ではこのTreadmillという手法を用いた並列GCをLisp1.5ベースの処理系に実装し，実装例の少ないこの手法の問題点などを解析し，評価を行なう．また，Treadmillに世代別GCの考え方を採り入れたGenerational Treadmillという手法を提案，実装し評価を行なうものである．

2 実時間GC

様々な実時間GCの手法の中から特にマークスイープ法を用いたのSnapShot GC[2]とPartial Marking GC[4]を紹介する．

2.1 SnapShot GC

マークスイープ法を用いてmutatorとcollectorを並行に処理を行なうとセルの参照関係に不都合が生じ，ゴミでないものがゴミとして回収されてしまう自体が起こる．

そこで，SnapShotGCでは**他のセルからの参照が切られた時はマークの対象にする**という約束を設けた．これによりGCが起動された時点で生きていたセルは必ずマークされることになる．あたかもGC起動の時点でスナップショットをとったかのようにGCを行なうことからSnapShot GC[2]と呼ばれる．

2.2 Partial Marking GC

SnapShot GCを改良したGCとしてPartial Marking GC [4]がある．SnapShot GCにはマークフェーズ中にでたゴミは回収できないという欠点がある．生成されたばかりのセルはゴミになりやすいという性質が世代別GCの研究からもわかっているため，この欠点はゴミの回収率に大きく影響する．そこで，Partial Marking GCでは，マークフェーズ中に生成されたセルだけを部分的にGCすることによりゴミセルの回収率を向上させた．つまり，Partial Marking GCはSnapShot GCに世代別の考え方を採り入れたGCということになる．この手法を採り入れることによって回収効率は2倍になることがわかっている[4]．

3 Treadmill

Treadmill[3]は1992年にBakerにより提案されたコピー法をベースとしたGCである．従来のコピー法を使用したGCはヒープ領域を2つ持っていたためリードバリアという処理系としては致命的な欠点を持っていた．しかしTreadmillではGC専用に双方向環状リストを使用することによってこの問題を解決している．

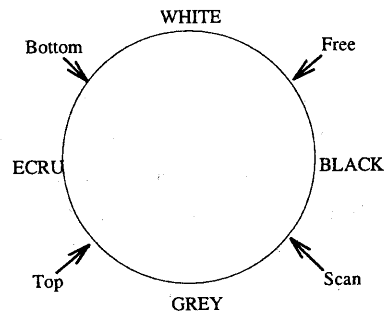

図1: Treadmillにおけるメモリ領域の状態

bottom, *free*, *top*, *scan*はシステムが保持するポインタで，これらをもちいてオブジェクトは以下のよう

に4色(white, black, grey, ecru)に色分けされる.

- **white**

  bottom と **free** 間にあるオブジェクトはまだプログラム中で利用されていないオブジェクト群(フリーリスト)を意味する.
- **black**

  free と **scan** 間にあるオブジェクトは現在使用中のオブジェクトを意味する.
- **grey**

  scan と **top** の間にあるオブジェクトは現在使用中ではあるが，そこからたどれるオブジェクトを調べていないつまりマーキング中であることを意味する.
- **ecru**

  top と **bottom** の間にあるオブジェクトはゴミかも知れないし，ゴミでないかも知れない. この間のオブジェクトは grey のオブジェクトがなくなった時にゴミであることが確定する.

collector は grey 領域を走査していく. もしもそこから指されているセルが ecru 領域にあるならば双方向リストを付け変えて grey 領域に移動する. 複写式インクリメンタル GC との一番の違いはここで**複写ではなく移動する**ことにある. この違いにより mutator にかかる負荷が大きく軽減される. つまり Treadmill は複写式インクリメンタル GC を改良した GC で複写式インクリメンタル GC の大きな欠点であったリードバリアを解消している.

4 Generational Treadmill

3で示したように Treadmill という手法は並列 GC アルゴリズムとして非常に有効であると考えられるが，SnapShot GC と同様に Snapping Phase 中に生成されたセルに関しては次回の GC までは回収されない(図2参照).

そこで，本研究では Treadmill という手法に世代別の考え方を採り入れ，GC を行なった後に GC 中に生成されたセルに関してのみ GC を行なう，Generational Treadmill という手法を提案する.

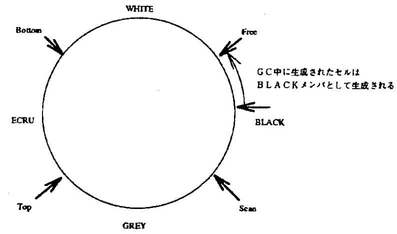

図 2: Treadmill における問題点

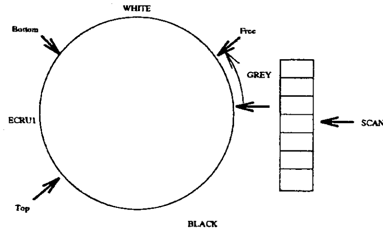

図 3: Generational Treadmill

各色の説明を以下に簡単に示す(図3参照).

- **white**

  bottom と **free** 間にあるセルはまだプログラム中で利用されていないセル群(フリーリスト)を意味する.
- **black**

  grey 領域から **top** の間にあるセルは現在使用中のセルを意味する.
- **ecru**

  top と **bottom** の間にあるセルはゴミかも知れないしゴミでないかも知れない. この間のセルは GC 用スタックが空になった時にゴミであることが確定する.
- **grey**

  この色のセルは GC 起動中に生成されたセルであり，ゴミであるかも知れないしゴミでないかもしれない.

Generational Treadmill は GC の Phase に2種類ある.

1. **Full Snapping Phase**

   root insertion を行なうことによって GC 用スタックに積まれたセルを collector は走査して行く. ス

タックが空になった時点でセル領域には black と ecru と grey の 3 色のセルが存在することになる．この時点で ecru を回収する．black のセルは black のままにしておく．

2. **Partial Snapping Phase**
root insertion を行なう．GC 用スタックに積まれたセルは Full Snapping Phase と同様のアルゴリズムで collector が走査して行く．collector は先ほど生き残った black のセルから先は走査しないので，結果的に Full Snapping Phase 中に生成された grey の領域にあるセルのみを black にすることになる．よって Partial Snapping Phase は短時間で終了することになる．終了したら grey のセルを回収し，black のセルは ecru に戻す．

上記の Full Snapping Phase と Partial Snapping Phase を繰り返すことによって GC を行なう．

5 実験結果及び考察

5.1 実験方法

本研究の実験においては Lisp1.5 ベースの処理系に Treadmill (以下 TMGC)，Generational Treadmill (以下 GTMGC)，Partial Marking GC (以下 PMGC) を実装し，SPARC Station 20/514(4CPU) 上においてセル数 30 万の条件でベンチマークプログラムを実行し次の点についてその時間の測定と比較を行なった．またそれぞれの collector はフリーセルの数がヒープ全体の 20%を切ったところで起動されるように抑制を行なっている．

- **ユーザタイム**
  リスト処理プロセスの総実行時間
- **GC ユーザタイム**
  GC プロセスの総実行時間
- **GC 回数**
  GC プロセスが GC を起動した回数

また，ベンチマークプログラムには次のものを利用した．

1. **フィボナッチ数列 (fib 30)**
2. **アッカーマン関数 (ack 3 8)**
3. **N-QUEEN (nqueen 11)**
4. **TAK 関数 (tak 27 18 9)**
5. **TARAI 関数 (tarai 16 8 4)**
6. **BIT (bit '(a b c d e f g h i j k))**
7. **BOYER**

5.2 結果及び考察

実時間 GC のアルゴリズムにとって重要なことはセルの枯渇を防ぐことと GC による停止時間をできる限り小さくすることであった．よって GC 1 回にかかる時間がなるべく少ない方が良い．たとえセルが枯渇したとしても 1 回の GC が少ない時間で終了すればすぐに mutator を起動することができ，セルが枯渇しそうな場合にも GC が少ない時間で終了すればセルの供給が早く行なわれるからである．

5.2.1 ユーザタイムと GC 回数の比較

Treadmill アルゴリズムと SnapShot アルゴリズムの違いはそれほど大きく現れなかった．しかし，TMGC と GTMGC の違いは顕著に現れた (表 1参照)．この違いはゴミセルの回収数によって現れたものと考えられる．TMGC と GTMGC に関して表 2をみると，GC に入る回数は GTMGC の方が多い．しかし，GTMGC には Full Snapping Phase と Partial Snapping Phase の 2 Phase が存在する．Partial Snapping Phase は Full Snapping Phase 中に生成されたセルにのみ GC を行なうのでコストは非常に小さくなる．root insertion などの処理が数ミリ秒で終わることと，Partial Snapping Phase において relink されるセルの数は極小数であることを考慮に入れれば，Full Snapping Phase と Partial Snapping Phase の 2 回の GC で TMGC の 1 回の GC とほぼ同コストと考えることができる．また回収するセルの個数は GTMGC の方が多いため，セルの消費率が同一であることを考えれば，GC を起動する回数は TMGC よりも少なくなる．よって結果的にユーザタイムに影響しているものと考えられる．

表 1: ユーザタイムの比較 (秒)

ベンチマーク	PMGC	TMGC	GTMGC
フィボナッチ数列	86.83	88.88	87.82
アッカーマン関数	130.17	134.72	131.71
N-QUEEN	86.11	80.13	79.04
TAK 関数	687.77	695.22	688.27
TARAI 関数	356.26	400.83	354.83
BIT	2.15	2.11	2.00
BOYER	29.73	31.94	31.46

表 2: GC 回数の比較 (回)

ベンチマーク	PMGC	TMGC	GTMGC
フィボナッチ数列	58	29	56
アッカーマン関数	120	66	124
N-QUEEN	60	32	64
TAK 関数	1000	585	1066
TARAI 関数	538	311	604
BIT	2	1	2
BOYER	34	20	38

5.2.2 GC ユーザタイムの比較

表 3からも明らかなように, PMGC よりも圧倒的に TMGC, GTMGC の GC ユーザタイムが少ない. これは, PMGC がマークスイープ法をベースにしたアルゴリズムであるのに対して, TMGC, PTMGC がコピー法をベースとしたアルゴリズムであるからと考えられる. マークスイープ法ベースの PMGC では GC を行なうために Mark Phase と Sweep Phase の 2 Phase 必要となるが, コピー法ベースの TMGC, GTMGC では Snapping Phase が回収作業も含んでいるので 1 Phase で終了する. Treadmill アルゴリズムにおいては Snapping にかかるコストが GC にかかるコストのほとんど全てであるので, 生きているセルが少ないアプリケーションでは特に有効性は大きい. 表 4を見ると PMGC の 1 回の GC にかかる時間はどのアプリケーションでもほぼ同じ位であるが, TMGC, GTMGC では生きているセルの数に応じて 1 回の GC にかかる時間が大きく変化する. 特に BIT などのような生きているセルが大量にあるようなアプリケーションでは, Snapping にかかるコストと mark, sweep にかかるコストが逆点し, Treadmill アルゴリズムの GC の方がコストが高くなる.

表 3: GC ユーザタイムの比較 (秒)

ベンチマーク	PMGC	TMGC	GTMGC
フィボナッチ数列	7.28	0.04	0.06
アッカーマン関数	17.10	0.74	0.82
N-QUEEN	8.41	2.29	2.18
TAK 関数	133.98	0.76	0.87
TARAI 関数	72.33	0.41	0.46
BIT	0.36	0.28	0.30
BOYER	5.37	3.82	3.56

表 4: 1 回の GC にかかる時間の比較 (秒/回)

ベンチマーク	PMGC	TMGC	GTMGC
フィボナッチ数列	0.1255	0.0014	0.0011
アッカーマン関数	0.1425	0.0112	0.0066
N-QUEEN	0.1402	0.0716	0.0341
TAK 関数	0.1340	0.0013	0.0008
TARAI 関数	0.1344	0.0013	0.0008
BIT	0.1800	0.2800	0.1500
BOYER	0.1580	0.1910	0.0937

5.2.3 アプリケーション実行時におけるセルの枯渇

PMGC は, SnapShot GC を改良した GC であることは 2.2で解説した. PMGC の collector は SnapShot GC の collector 2 つと同程度の効率でゴミ処理を行なうことができるが, それでも BIT などのように大量のセルを消費しつつセルの生存率が高いようなアプリケーションでは collector のフリーセルの供給が追い付かず mutator が停止してしまう状況が見られる. 本実験においても実際に BIT を動かした場合に PMGC と TMGC においてセルの枯渇が生じてしまった. GTMGC は枯渇することはなかった. 本実験では collector の動作を抑制するためのパラメータとしてフリーセルのヒープに占める割合 (以下 CI パラメータと呼ぶ) を基準にして残りのフリーリストが 20 切った時点で collector を起動するようにしてある. 既述の実験結果とは別に BIT を実行する際にどの程度まで CI パラメータを下げることができるか実験を行なった (表 5参照). この

結果から見てもわかるように実際にセルの枯渇に対して Generational Treadmill は非常に強いということがわかる.

表 5: GC 起動の限界点 (%)

	CI パラメータ
PMGC	24
TMGC	21
GTMGC	10

5.2.4 実験の考察と問題点の解決

Treadmill アルゴリズムを用いた場合の問題点として，セルの生存率が高い場合の GC によるオーバヘッドが大きいということが挙げられる．これを解決するためには，何回か GC を生き残ったセルは他の領域に移し通常の GC の対象外にする手法 (Tenuring)[5] をとることによってより一層の効率の向上が望める.

また，コピー法をベースとした GC の手法であるためアプリケーションの性質によって 1 回の GC にかかる時間が変動する．特にセルの生存率は大きくシステムの効率に影響し，生存率が高ければ高いほど効率は悪くなる．Tenuring との併用によって改善されることが考えられるが，生存しているセルが多いアプリケーションではなるべく早く GC を起動し，生存しているセルが少ないアプリケーションではぎりぎり迄 GC を起動しなくても良い．生存率の低いアプリケーションでは Treadmill アルゴリズムを使用している場合は 1 回の GC にかかる時間が非常に短時間なので，通常のマークスイープ法ベースの GC よりも GC の起動回数を削減することができる．このように，様々なアプリケーションに対応したシステムを実現するにあたっては GC の起動のタイミングをアプリケーションによって自動的に変動させるような機構が望まれる.

6 結論及び今後の展望

6.1 結論

本研究においては，Treadmill アルゴリズムを使用した並列 GC を Lisp システム上に実装し，並列 GC ではあまり有効ではないとされてきたコピー法ベースのアルゴリズムで，リードバリアを解消したシステムを実現した．また Treadmill アルゴリズムを使用した Lisp システムは，特に生きているセルが多くないアプリケーションでは非常に有効であるということを示した．さらに，世代別 GC の考え方を採り入れた Generational Treadmill という手法を提案しその有効性についても示した.

6.2 今後の展望

5.2.4にも示した通り，現状のシステムの改善点として

- **Tenuring の併用**
- **アプリケーションの性質に対応した動的な GC 起動の機構**

が挙げられる．今後は，上記の点について研究を進め，様々なアプリケーションに対応したより効率良い実時間処理を目指す予定である.

参考文献

[1] Baker H.G., List Processing in Real Time on a Serial Computer. Communications of the ACM, Vol. 21, No. 4 1978.

[2] Yuasa T. Real Time Carbage Collection on General Purpose Machines. Journal of Systems and Software, Vol. 11, pp. 181–198, 1990.

[3] Baker H.G. Jr. The Treadmill : Realtime Garbage Collection without Motion Sickness, ACM SIGPLAN Notices, Vol. 27, No. 3, pp. 66–70, 1992.

[4] Tanaka, Y., Matsui, S., Maeda, A. and Nakanishi, M. Partial Marking GC Proceedings of International Conference on Third Joint International Conference on Vector and Parallel Processing (CONPAR 94 – VAPP VI), pp. 337–348 Sep. 1994.

[5] Eiko Tanaka, Yoshio Tanaka, and Masakazu Nakanishi. A theoretical analysis of advancement threshold in generational garbage collection. In *APPLIED INFORMATICS, Proceedings of the Thirteenth IASTED International Conference*, No. 230-119, pp. 378–381. IASTED, February 1995.