Snapshot型並列GCにおけるルート挿入時間の削減

岩井輝男† 中西正和††

並列GCは，リスト処理の実時間処理（無停止処理）の実現が可能なGC方法である．snapshot型アルゴリズムの並列GCでは，ルート挿入時点で生きていたオブジェクトのsnapshotをとるため，ルート挿入時にはmutatorとcollectorとで同期を取り，mutatorのリスト処理を一時中断してmutatorのルート挿入を行わなければならない．1回の停止時間をできるだけ小さくするためには，同期の時間を削減するか，mutatorのルート挿入時間を削減するかのいずれかである．同期の時間はハードウェアとオペレーティングシステムの機能に関係し，停止時間を減少させる事が難しい．我々はsnapshotを取るためのルート挿入は，mutatorがルートを書き換えるまで行う必要がない事に着目した．書き換えの起こったルートの含まれるページだけのルート挿入を行うことで，一般的なsnapshot型並列GCのルート挿入によるmutatorの停止時間を減らすアルゴリズムを提案する．並列Lispへの拡張性を考えこの並列GCの手法を用いて実装し実験した結果，GC時のルート挿入による停止時間が一定時間内に抑えられることが確認された．この結果は，実時間GCの実用化に大きく貢献すると考えられる．

Reduction of Pause Time due to Snapshot Parallel GC

TERUO IWAI† and MASAKAZU NAKANISHI††

Parallel garbage collection enables list processing in realtime(non stop processing). The garbage collection (GC) of snapshot algorithm is one of parallel GC. It is necessary for this GC to use the mutual exclusion between mutator and collector for taking the snapshot at root insertion, and to pause temporarily the mutator, and to make mutator do root insertion. For one pause time is as short as possible, it is necessary to shorten each the time of mutual exclusion or root insertion. The time of the mutual exclusion depends on hardwire and operating system, and it is difficult to shorten the pause time. We note that the root insertion is not necessary until the mutator rewrites root area. We propose the method of decreasing pause time that approaches that only pages that is rewritten are copied. For the expansion to parallel Lisp we implement this method, experiment, and obtain that the pause time of mutator is less than certain constant time. The GC with this method contributes to practical use realtime GC.

1. まえがき

低レベルの言語では，プログラムで明示的にメモリの割り当てと解放，再利用を行う必要がある．Lisp言語などの高級言語では，実行時にメモリの領域の割り当てと解放を暗黙のうちに処理を行うようになっている．これを行うために解放すべきかどうかの判断を行い，解放できるものは再利用する必要がある．この処理をガーベッジコレクションと呼ぶ（以下GC）．プログラムが実行しているいる時に実行を一時停止してその間にGCを行う，という停止型GC法が考えられてきた．この手法では，GCの処理の時間は必ずプログラムが停止していることになる．プロセッサの性能の向上によりGCのための停止時間は減ることになるが，アプリケーションが大きくなり，より多くのメモリを使用するような場合などでは停止時間は大きくなる．このためインタラクティブなシステムや，ハードウェアを直接制御するような実時間性が必要な場合には不適である．

これに対処するためにアプリケーションとGCを同時に行うことで停止時間を減らす方法が考えられ，並列GC法が提案されてきた．並列GCのアルゴリズムで最も多く採用されているのは，snapshot型GC[1]である．メモリ領域のイメージであるsnapshotをと

† 慶應義塾大学大学院理工学研究科
Graduate School of Science and Technology, Keio University
†† 慶應義塾大学理工学部情報工学科
Department of Information and Computer Science, Faculty of Science and Technology, Keio University

る時間だけアプリケーションを停止させ，とり終わるとアプリケーションを再開させ，再開後に GC を行うという方式である．snapshot をとる時間は，GC の処理全体と比較して一般的に小さいのでアプリケーションの停止時間を減らすことになる．

リスト構造を構成するオブジェクトはオブジェクト間の複数のリンク（ポインタ）を含んでいる．リストの出発点となる特定のオブジェクトへのポインタの集合をルートと呼ぶ．ルートから到達可能なオブジェクトは生きているオブジェクトであり，到達不可能なオブジェクトはごみオブジェクトと見なす．

snapshot をとる場合にはヒープ全体[2]をとる場合と，ルートのみをとる場合がある．基本的にルートはヒープ全体の領域と比較して小さい．よって snapshot をとるための効率を考えた場合，ルートの snapshot をとる方法の方が snapshot 型並列 GC では一般的である．

ルートとなる領域はスタックなどである．スタックの大きさは実行時に変動する．よって snapshot をとる時はスタックの大きさが少ない時に行うと停止時間も少ないが，実際にはスタックの大きさがいつ小さくなるのか判断できないために，スタックの大きさを考えずに snapshot をとらざるを得ない．

本研究では，ルートとなるスタックの大きさが大きい場合でも，ある一定の停止時間内で snapshot をとるための手法を提案する．これは汎用計算機には一般的に利用されている Memory Management Unit(MMU) の機能を使った手法である．この手法を並列 Lisp でもこの並列 GC 法が適用できるように手法を考え，この手法を実装し，停止時間について測定を行う．

2. snapshot 型並列 GC

マークスイープ法を並列化する方法として，ルートの snapshot をとり（これをルート挿入と呼ぶ）その snapshot からたどることが可能なオブジェクトに印付けを行う．この時に印が付かなかったオブジェクトを回収するという方式がある．この方式では mutator を停止する時間は，ルート挿入を行う時間だけで十分でありこの時間以外は処理を行うことが可能である[1]．ルート挿入中にリスト処理を行うと，snapshot をとることが保証されないために GC の不整合を起こす．このために必ず停止する必要がある．

3. ルート挿入の時間

ルート挿入の時間は，mutator のリスト処理を停止させてルート挿入を行いリスト処理に戻るまでの時間となる．よって停止時間は，scheduler と同期を取る時間とルート挿入の時間から成る．

mutator はルート挿入の始まりや終りを scheduler に必ず通知する必要がある．通知するためには，プロセス間（thread 間）で同期を取る必要がある．同期の方式などはいろいろ挙げられている．thread 間で共有しているメモリに対して spin ロックを使った busy-wait 方式や，lock のキューを使った同期の方式などについて比較されている[3]．また，オペレーティングシステムの機能を使った方式なども考えられ，同時に同期を取るプロセッサの数や，ハードウェアのメモリのバンド幅などのハードウェアに大きく依存する部分が大きい．同期の時間はこれらの条件が同じであれば，ルート挿入の方式に関係なく固定時間となる．よって，ルート挿入の時間についてのみ考える．

ルート挿入において，ルートを分割してルート挿入を行う方式について考える．ルートは，レジスタを含む固定領域とスタック（動的領域）に分けることが可能である．この中ですぐに書き換えられるかまたはランダムに書き換えられる可能性があるのは固定領域であるのに対して，スタックは LIFO に従った読み書きがされるために全領域がすぐに書き換えられたりランダムに読み書きされることはないものと考えられる．よって，スタックについてはある程度規則的なアクセスをするのでこれを考慮して mutator がリスト処理中にルートを書き換える前にルート挿入を行うことで snapshot をとることには影響しない．この事に着目すると snapshot をどの時点にとるかによって次の 2 つの方法が考えられる．

- ルート挿入中の snapshot（bottom up 法）
- snapshot 後のルート挿入（top down 法）

この 2 つの方法について説明を行う．

3.1 bottom up 法

bottom up 法[4]は，collector が先に mutator の動的領域のルート挿入を行う．図 1 のように，mutator は実行時にスタックポインタを上下に移動させる．この方法ではスタックポインタを下側に移動して，ルート挿入を既に行った領域に対して mutator が書き込みを行った時，mutator がレジスタを含む固定領域のルート挿入を行う方式である．この手法の場合，mutator が既にルート挿入を行った部分に書き込んだ直後のルートが snapshot としてルート挿入を行ったことになる．mutator はスタックに対して，LIFO のデータのやり取りを行うために，スタックのトップからすぐにスタックの底にアクセスするようなことはあり得ないと考えられる．この事から，collector は図 1 の

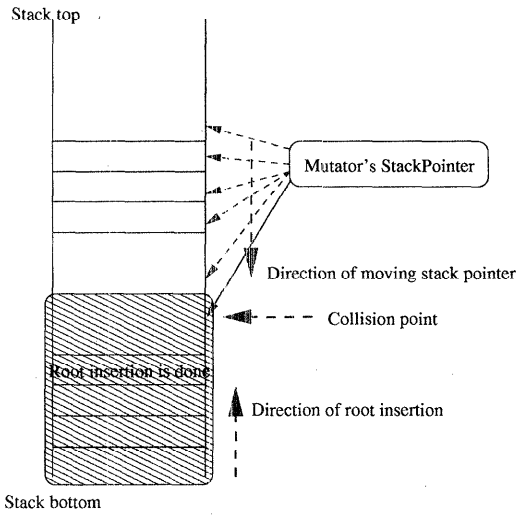


図 1 bottom up 法
Fig. 1 Bottom up method.

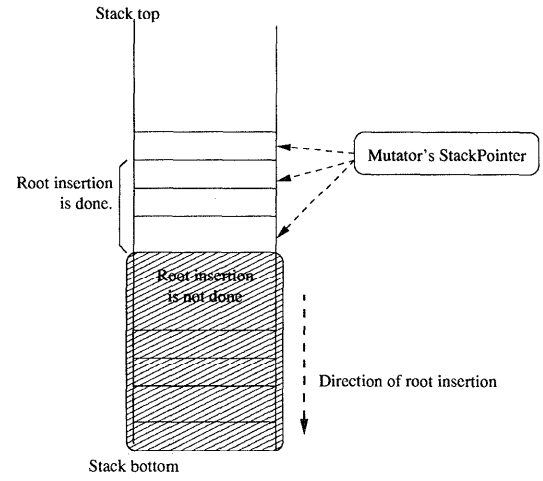


図 2 top down 法
Fig. 2 Top down method.

ようにスタックの底からルート挿入を始める.

3.2 top down 法

top down 法は mutator のリスト処理を中断して, レジスタなどを含む固定領域とスタックポインタを含む領域のページのルート挿入を行って, リスト処理に復帰する. その後で図 2 のように scheduler の thread がまだ行っていないスタックの部分のルート挿入をスタックトップ側から底側に行う. mutator のスタックポインタはリスト処理実行中に上下に移動する. スタックポインタが移動して既にルート挿入が終わった部分に対して書き込みを行っても snapshot をとることに関しては問題はない.

ただしルート挿入がまだ終わっていない動的な領域の一部を mutator が書き換えようとした場合には, その領域を含む部分を mutator 自身がルート挿入を行い, リスト処理に復帰する. すべての領域をルート挿入が終了したかどうか scheduler が判断して次のフェーズに移る. この方式では場合によって, 全てのルート挿入を行うために mutator は複数回停止する可能性があると考えられる.

mutator は LIFO に従ってスタックの読み書きを行う. mutator がスタックを書き換える前にできるだけ多くのページのルート挿入を scheduler が行うためには, 図 2 のようにスタックの上から下に向かってスタックのルート挿入を行えばよい. このようにすることで scheduler のルート挿入の方が速ければ, mutator 自らスタックのルート挿入を行う回数は最初を除いて起きることはない.

3.3 並列処理への適用

mutator の数を複数にした場合について考える. bottom up 法では mutator の書き込みと, scheduler のルート挿入を行った部分がぶつかった時に mutator のレジスタなどのルート挿入を行うが, 1 つの mutator 以外はルート挿入はまだ終了していないので他のすべての mutator のルート挿入を行う必要がある. この場合 2 つの手法が考えられる.

1 つはその時点ですべての mutator のスタックをルート挿入を行う手法である. この場合スタックをあまり使わない mutator が 1 つで他は多く使う場合には停止時間が多くなってしまい, この手法の意味がなくなる.

もう 1 つは, スタックでまだルート挿入を行っていない領域は 1 ページ毎に top down 法を用いてルート挿入する方法である. この方法であれば停止時間は前者の方式と比較して少くなる.

また bottom up 法ではスタックを分割して scheduler や複数の collector を使ってルート挿入する場合, ルートの大きさが決定しない前からルート挿入を行う. このため snapshot 先の大きさが決定できない. この事からスタックの大きさの最大値を予想して領域の確保を行うか、あるいはルート挿入毎にある単位毎に領域を確保する必要がある. このために, ルート挿入を行う全 thread 間での同期を取ってルート挿入する必要性がある.

以上の事から並列処理の適応性を考えた場合には top down 法のほうが適応しやすいと考えられる. よって本研究では top down 法を用いる.

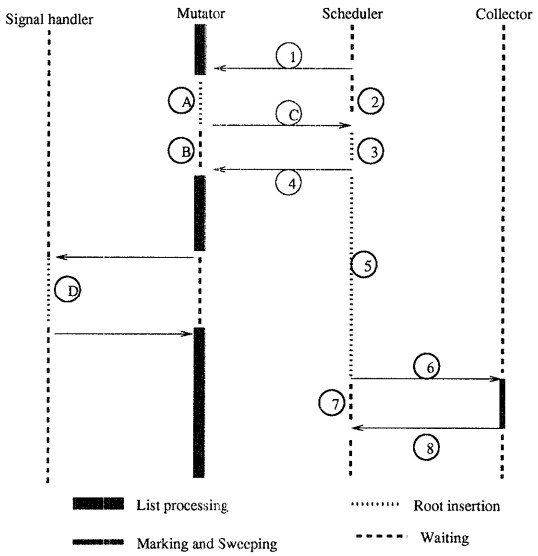


図 3 改良後の GC アルゴリズム
Fig. 3 Improved GC algorithm.

4. ルートの分割ルート挿入

top down 法における snapshot をとる方式について，図 3 に示す．

4.1 mutator

scheduler からルート挿入要求（図 3 の①）が来た時，現在使っているスタックのページ（スタックポインタがあるページ）とレジスタなどの固定領域のみのルート挿入を行う（図 3 のⒶ）．このルート挿入が終了したことを scheduler に報告する（図 3 のⒸ）．この後，scheduler は複数の mutator 間で共有しているルート（グローバルルート）をルート挿入を行う（図 3 の③）．そして，mutator は scheduler からリスト処理再開の合図（図 3 の④）が来るまで待つ（図 3 のⒷ）．mutator はリスト処理を再開し，その間に mutator がルート挿入を行っていない残りのスタックを scheduler がルート挿入を行う．scheduler がまだルート挿入を行っていないスタックの一部に mutator が書き込みを行った場合 mutator は一時処理を中断し，signal handler を実行する．signal handler では，ルート挿入がされていない部分のページのみのルート挿入を行う（図 3 のⒹ）．mutator は signal handler から復帰し書き込みに失敗した部分に再度書き込みを行って通常のリスト処理に戻る．

4.2 scheduler

scheduler は，mutator と collector の間の同期を取るために以下の動作を繰り返す．

① mutator へのルート挿入要求発信
② mutator からの現在使用中のスタックのページとレジスタのルート挿入完了の合図待ち
③ グローバルルート挿入
④ mutator へのルート挿入完了の合図発信
⑤ まだルート挿入を行っていないルートをページ毎にルート挿入を行い，書き込み保護を外す
⑥ collector への印付け，回収開始の合図発信
⑦ collector からの印付け，回収完了の合図待ち
⑧ collector からの印付け，回収完了の合図受信

4.3 collector

collector は scheduler から印付け，および回収の開始合図が来た後で（図 3 の⑥），印付けと回収の処理を行い，scheduler に終了の合図を行う（図 3 の⑧）．

4.4 top down 法の分割ルート挿入のアルゴリズム

この手法を用いたルート挿入のアルゴリズムを図 4 に示す．scheduler のアルゴリズムはルート挿入部分についてのみ示す．図 4 のプロシージャ signal_handler のパラメータの fail_address は mutator が書き込みに失敗したアドレスを表す．

まず，この GC の手法の正当性について述べる．既に snapshot 型 GC は GC について正当であることが証明されているので，この手法が snapshot 型 GC と同じ手法であることを証明すれば，この GC の手法が正当であることを証明したことになる．

これを証明するために 2 つのことを証明する必要がある．snapshot 時点のルートのページ数とルート挿入された先のページ数は同じであることと，snapshot 時点のルートとルート挿入された先の snapshot が一致することである．

図 4 により，scheduler はすべてのスタックのページがルート挿入されているかどうかの判断を行っている．よって，スタックのルート挿入されているページ数は同じであると言える．

ルート挿入によって作られたものは snapshot 時点のルートと一致することをを証明する．スタックへの書き込みを行うのは mutator である．mutator が書き込みを行う時は 2 つの場合が考えられる．既に scheduler がルート挿入を行って書き込み保護が外されている場合と，図 4 の signal_handler を実行してから mutator が書き込みを行う場合である．どちらの場合も，snapshot 時点でのスタックをルート挿入した後で書き込みを行うことになる．よってルート挿入を行ったものは snapshot 時点のルートと一致する．

以上により，この分割ルート挿入は snapshot をと

```
procedure mutator のルート挿入;
begin
  固定領域とスタックの一部をルート挿入を行う;
  ルート挿入していないスタックに書き込み保護をかける;
end;
procedure signal_handler(fail_address)
begin
  if fail_address のページはルート挿入中である then
     while fail_address のページはルート挿入中である do
  else
    begin
      fail_address のページのルート挿入する;
      fail_address のページの書き込み保護を外す;
    end
end;
procedure scheduler のルート挿入;
begin
  foreach a_page in ルート挿入の対象となるスタックのすべてのページ
  do
    if a_page はルート挿入されている途中でない and a_page はまだルート挿入されていない then
    begin
      a_page をルート挿入する;
      a_page の書き込み保護を外す;
    end
end;
```

図 4 分割ルート挿入のアルゴリズム
Fig. 4 Algorithm of divided root insertion.

ることができていることが示される．よって本手法はGCについて正当である．

4.5 実 装 法

プロセッサのMMUの機能を使って1ページ単位に書き込み保護をかけることが可能である．保護された領域に書き込みを行うとシグナルハンドラが実行される．この機能を使ってルート挿入をページ毎に分割して行う並列GCを実現する．

実際には，実装に用いたオペレーティングシステムでは，ページ毎に書き込み保護を設定することが可能であり，さらに書き込み保護されたページに書き込みを行うとsegmentation fault☆のシグナルハンドラが呼び出される．このシグナルハンドラには書き込みに失敗したアドレス（ページ）などが渡される．シグナルハンドラ内部で書き込みに失敗したページだけルート挿入を行い，書き込み保護を外す．schedulerがそのページをルート挿入中である場合は終わるまで待つ．どちらの場合もルート挿入を行った後でシグナルハンドラから復帰する．mutatorはこの後，書き込みに失敗した命令を再実行し，mutatorのリスト処理には何も起きなかったように処理に復帰する．

この手法ではMMUの機能を使っているので，スタックを減らす処理毎に別のページへポインタが移動するかどうかの判断や，その時にそのページがルート挿入されているかどうかの判断を行う必要はない．すなわち，スタックフレームの内容の解析は不要である．

実際に最初にmutator自身でルート挿入するのは，固定領域と現在のスタックポインタがある領域の1ページとレジスタであるが，これでは続けてmutatorがルート挿入を行う場合があり得る．

例えば，1つのスタックフレームが2つのページに股がって配置されているような場合，スタックフレーム内部ではランダムに読み書きされる可能性が高いた

☆ オペレーティングシステムがUNIXに属する場合，このシグナルがあるのが一般的である

め，股がっている2つのページに対して連続して書き込むことが起きるとmutatorは連続してルート挿入を行う．これを避けるために，1スタックフレーム分を含むスタックのページのルート挿入を行う．これによってスタックポインタがページの切目の近くの場合でも，大域的な脱出がなければ連続してmutatorがルート挿入することはない．

4.6 ルート挿入の競合

mutatorがスタックを減らしていく前にschedulerがルート挿入を行えばmutatorは最初のルート挿入を除いて行う必要はない．またmutatorのスタックを減らす方が速ければmutator自身でルート挿入を行う必要がある．よって2つのtheadのルート挿入の速度の差でmutatorの停止回数が変わる．mutatorの処理で再帰のようにスタックが減るだけのような場合とschedulerのルート挿入の速度を比較すると，schedulerの場合は1ページのルート挿入が終わる毎にそのページの書き込み保護を外す必要がある．書き込み保護を外すにはオペレーティングシステムのシステムコールを呼び出す必要がある．このコストは1ページのルート挿入の時間と比較して少ないコスト☆では済まない．このためmutatorが関数から復帰するような戻り値をスタックに書き込んで単にスタックポインタを戻すような場合にはschedulerのルート挿入の速度のほうが遅いと考えられる．

schedulerとmutatorの同期を取る部分では，schedulerが中心で同期の処理を行っている．schedulerがmutatorに対して同期を取る場合，非同期のメッセージ送信の形で行っているために，mutatorがリスト処理への復帰するよりも，schedulerのルート挿入のほうが先に行うことになる．この時間の差の間にmutatorがスタックを書き換えるよりも前にschedulerがスタックのルート挿入を部分的に行うことが可能である．

5. 関連研究

GCの時間をできるだけ少なくするという研究には，並列GCの研究がある．並列GCはほとんどがsnapshot型のマークスイープ法である．snapshot型のマークスイープ法でハードウェアのMMUの機能を使って，全領域の複写を行う方式が提案されている[2]．この方式では，広い範囲に書き込みを行う時には，書き込みを行ったページを連続して複写することになる．また，この方式は書き込みが行われるまで複写しない方式である．この研究の結果としては，複写に時間がかかり回収が間に合わないという結果が得られている．これは，ヒープ領域全体の複写を行うことで時間がかかっていると考えられる．Appelら[5]はsnapshot型の並列GCではなく，複写型の並列GCを提案している．この手法もMMUの機能を使って要求毎にその1ページ分の複写を行う方式である．この場合書き込み保護以外に，mutatorだけからは読み込み保護も行う必要があり，実装法が複雑で，あまり一般的ではない．

並列GCの意味の1つとして停止時間が少ないことがあげられるが，今までの研究では停止時間を少なくすることについては考えられていなく，アルゴリズムの提案や実行時間全体の効率向上に過ぎない．

6. 実験および評価

本手法では，mutatorのスタックが急に小さくなるとschedulerのルート挿入が間に合わず，mutatorがルート挿入を行う場合がある．4.6節に述べたように，schedulerのルート挿入を先に行うために，スタックの大きさがどの程度ならば，最初以外のmutatorのルート挿入を行わずにschedulerのルート挿入だけでルート挿入を終了させることが可能であるかについて調べる．さらに従来の手法と本手法を用いた場合のmutatorの停止時間について実験を行い評価を行う．

本研究で提案している方式についてEUS Lisp[6)~11)]に実装を行い，以上の2つの実験を行った．実験を行った環境は，共有メモリ型の並列計算機SPARC Center 2000上で行った．プロセッサ数は16個で，物理メモリは，768Mバイトである．1つのthreadを1つのCPUに割り当てて実験を行った．

6.1 mutatorのルート挿入の回数

本実験の目的は，mutatorのスタックを減らす速度が遅ければmutatorが最初のルート挿入を除いてルート挿入を行うことはないが，スタックを減らす速度が最も速い場合は，何回mutatorはルート挿入を行う必要があるかについて調べる．ここでは，mutatorのルート挿入の最大回数を調べる．

実験に用いたプログラムは，Lisp言語で記述し，1～900までの和を再帰を使って求める処理を1000回繰り返すプログラムである．繰り返すことでsnapshot時のスタックの様々な大きさに対するルート挿入の回数を得ることが可能となる．これは最もスタックを減らす速度が速いプログラムであり，900までの和を計算するまで自己再帰を行った後，関数から復帰する毎にその結果をスタックに書き込みを行って復帰するまでを繰り返すプログラムである．このプログラムは上記のこのプログラムをコンパイルして実行を行うが，

☆ プロセッサのクロックでは数千サイクル程度である．

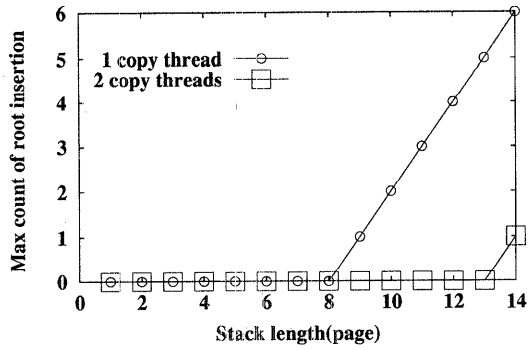


図 5 mutator によるページの最大ルート挿入回数
Fig. 5 Max copy count of page by mutator.

普通の場合コンパイルを行ったプログラムコードは再帰ではなく，ループに変換する．この実験の場合はループに変換しないようにコンパイルを行ったプログラムコードを使用した．また，このプログラムでは GC を必要とするほどオブジェクトを使用しないが，ルート挿入の時間を調べるために GC を行うように実験を行った．使用したオブジェクトの数を判断して scheduler は GC を起動するが，このアプリケーションでは起動されるほどオブジェクトを使用しない．よって，使用したオブジェクトの数を見ないように scheduler を変更して GC を起動するように行った．

この実験では snapshot の時点がスタックが増加する時なのか，減少する時なのかどうかの判断を行うことができないために，mutator が行ったスタックのルート挿入の回数が最大何回あったかどうかについて調べた．よって図 **5** の結果は最大値をとっている．実行した回数はこのプログラムを 100 回程度実行したときの結果である．snapshot 時点のルートのスタックの大きさ（ページサイズ）を横軸として，縦軸は mutator のルート挿入を行ったページの数の最大値である．ただし，最初のルート挿入は回数に含んでいない．

scheduler だけでルート挿入を行う場合は図 5 の 1 copy thread のグラフであり，scheduler とさらに collector もルート挿入を行う場合は，2 copy threads のグラフに示す．この結果から 1 つの thread だけでルート挿入を行う場合ルートの大きさが 8 ページ以下の場合のみ最初のルートの停止以外にルート挿入行っていない．8 ページを越える場合には 1 ページ増えることに停止回数が 1 回増えている．2 つの thread を用いてルート挿入を行った場合には，13 ページまでには最初のルート挿入のための停止以外に停止していない．2 つの thread に分けた場合はルート挿入を行うページを，ページ毎にインターリーブに分けてルート挿入を行う方法である．

scheduler がスタックをルート挿入する速度と最も早く mutator がスタックを減らす（関数からの復帰）速度は mutator のほうが速いことから，scheduler のルート挿入のほうが間に合わない可能性がある．しかし scheduler の方が先にルート挿入を行うために，mutator よりも有利であり，図 5 の結果からこのような事はないことが分かる．また，collector も scheduler とルート挿入することで scheduler だけでルート挿入を行うよりも mutator の停止回数を減らすことが可能であることが得られた．

6.2 mutator の停止時間

snapshot 時点の最初のルート挿入を含む mutator の停止時間について，1 停止回数あたりの平均値，最大，最小時間ついて調べる．また，本手法を用いない従来の手法についての停止時間についても実験を行う．実験に使用したプログラムは先に説明した 1 ～ 900 までの和を求めるプログラムである．

図 **6** は停止時間の結果である．グラフは Collective が本手法を用いない場合の snapshot 型並列 GC の場合であり，Division は本手法を用いた場合である．最大値と最小値の範囲を示し平均値でグラフの線を引いてある．この結果から従来の方式ではスタックの大きさによって 1 回の停止時間が比例して延びていくのに対して本手法を用いた場合は 1 回の停止時間は 1m 秒程度になっている．この結果と先の図 5 から 1 回の GC の合計停止時間を考えた場合，8 ページ以下の場合には 1 回の停止時間で 1m 秒となり，8 ページを越える場合には，1 ページ増す毎に 1m 秒だけ停止時間が増えるが，停止時間の合計は従来の方式と比較して小さいことが分かる．

実験で用いたプログラムは特別なプログラムであるが，ルート挿入に関してはルートの大きさだけに関係し，アプリケーションには依存しない．

7. むすび

他に実験を行ったプログラムとしては，論理証明器の MALL の証明 [12),13)] のプログラムがある．これについて実験を行ったところ，スタックの大きさは 10 ページまでになっているが，mutator は最初の 1 回だけの停止回数しかなかった．

スタックのルート挿入を行う thread を 2 つ以上にした場合にはハードウェアに依存し，メモリのバンド幅などに影響する．メモリバンド幅が広ければルート挿入を行う thread の数に比例してページをルート挿

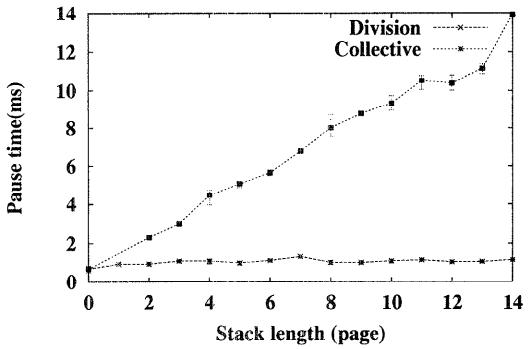


図 6 1 回の停止時間
Fig. 6 Pause time.

入を行うことができると考えられるが，本システムでは 2 倍までは達していない．

本研究では並列 GC の目的の 1 つである停止時間が小さいことがあげられるが，この並列 GC を一般的な汎用計算機に実装し時間を求めた．しかし，停止時間がスタックの大きさにほぼ比例する結果が得られた．さらに停止時間を減らすために，スタックを分割して mutator 以外の thread が並列にルート挿入するという手法を提案した．この手法は複数の mutator にも同様に対応可能である．この手法を用いることで snapshot の対象となるスタックの大きさに関係なく従来の 1 ページ分の停止時間程度にすることが可能であることを示した．scheduler のルート挿入が mutator の処理に間に合わない時にはルート挿入を行う thread を複数にすることで可能であると示した．mutator を複数にした場合において，停止時間がどの程度増えるかについても実験を行いこの手法の評価を行う必要があると考えられる．

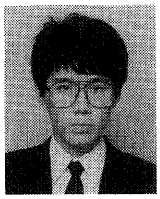
本システムは SPARC Center 2000 というマルチプロセッサのワークステーションで実験を行ったが，一般的にほとんどの汎用ワークステーションでは MMU を使っているため，snapshot 型 GC にこの方式を実装することは可能であり，我々の提案するアルゴリズムは，オブジェクト指向言語などにも適応可能である．

謝辞 本研究を行うために並列計算機を使わせて頂き、電子総合研究所の松井俊浩様に感謝致します。さらにこの計算機を用いて本研究の実験を行ったために他の方にご迷惑をおかけしたことをお詫び致します．

参 考 文 献

1) Yuasa, T.: Real-Time Garbage Collection on General-Purpose Machines, *Journal of Systems and Software*, Vol.11, No.3, pp.181–198 (1990).
2) Furusou, S., Matsuoka, S. and Yonezawa, A.: Parallel Conservative Garbage Collection with Fast Allocation, *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems* (Wilson, P. R. and Hayes, B.(eds.)) (1991).
3) Jang Seung-Ju, K. G.-Y.: Spin-Block Synchronization Algorithm in the Shared Memory Multiprocessor System, *ACM Operating Review*, Vol. 28, pp. 5–30 (1994).
4) 近藤豪, 中西正和: 実時間ごみ集めにおけるルート挿入の効率化, 情報処理学会研究報告, No. 16, 情報処理学会 (1997).
5) Appel, A.W., Ellis, J.R. and Li., K.: Real-time Concurrent Colection on Stock Multiprocessors, *SIGPLAN'88 Conference of Programming Language Design and Implementation* (1988).
6) 松井俊浩, 稲葉雅幸: EusLisp: オブジェクト指向に基づく Lisp の実現と幾何モデラへの応用, 情報処理学会記号処理研究会, Vol. 89, No. 50 (1989).
7) 松井俊浩, 原功: EusLisp version 8.0 Reference Manual, Technical Report ETL–TR–95–19, 電子技術総合研究所研究速報 (1995).
8) Matui, T. and Inaba, M.: EusLisp: an Object-Based Implementation of Lisp, *Journal of Information Processing*, Vol.13, No.3, pp.327–338 (1990).
9) 松井俊浩, 関口智嗣: マルチスレッドを用いた並列 EusLisp の設計と実現, 情報処理学会論文誌, Vol. 36, No. 8 (1995).
10) 松井俊浩, 関口智嗣: マルチスレッド EusLisp の分割型メモリ管理手法, 情報処理学会プログラミング研究会, Vol. 3, No. 3, pp. 9–14 (1995).
11) 松井俊浩: オブジェクト指向型モデルに基づくロボットプログラミングシステム, 技術報告 第 926 号, 電子技術総合研究所研究報告 (1991).
12) 竹内外史: 線形論理入門, 日本評論社 (1995).
13) 山口文彦: MALL の証明器, *URL http://www.nak.ics.keio.ac.jp/groups/li/.*

(平成 10 年 10 月 16 日受付)
(平成 10 年 12 月 16 日採録)



岩井 輝男（正会員）
平成 8 年慶應義塾大学大学院理工学研究科博士課程単位取得退学．同年より同大学研究生．Lisp に興味をもち，並列 Lisp，並列処理の研究を行う．

中西　正和（正会員）

昭和41年慶應義塾大学工学部卒業．昭和44年同大学工学部助手．平成1年同大学理工学部教授．工学博士．昭和42年日本初の実用Lisp処理系を作成．以後，記号処理言語，人工知能用言語等の研究に従事．昭和57年Lispマシン SYNAPSEの開発など．