記 号 処 理 61－5
（1991. 9. 13）

セル消費監視プロセスの設置による Generation Scavenging GC

高岡詠子　中西正和
慶應義塾大学

GC に object の寿命の概念を取り入れた Generation Scavenging GC は一回の GC によって生じるポーズタイムを大幅に短縮した手法である。この手法では object は生成されてから通常の GC をある一定の回数以上生き残ると寿命の長い object とみなされ、通常の GC の対象外となるが、その回数（*Advancement Threshold*）をいくつに設定するかという問題が残される。本稿では、Advancement Threshold の数字の持つ意味の重みがアプリケーションによって変わってしまうことを指摘し、アプリケーションのセル消費に合わせた効率の良いガーベッジコレクションを行なう一方法を提案し、現在行なっている実験について報告する。

Generation Scavenging GC with a process monitoring the rates of cell use

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

Generation Scavenging GC, which is a variety of a garbage collection, divides objects into generations according to their ages. As there will be little live data to copy, the cost of this garbage collection is low. In this algorithm, when an object survives some number of scavenges, it is advanced to the older generation. We call this advancement *tenuring*. If many *tenured* objects die, much space will be wasted. We propose an efficient Garbage Collection that is adaptive for the rate of cell use. We report experiments on the rates of cell use of many application.

1 はじめに

リスト処理言語にとって、Garbage Collection（以下GC）をはじめとする自動的なメモリの管理は不可欠である。リスト処理言語の発達にともない、GCの研究が盛んに進められている。

GCにobjectの寿命の概念を取り入れたGeneration Scavenging GC[Ung84]は通常のGCを寿命の短いobjectに限定して行ない、寿命の長いobjectにかかるGCの時間を削減することによって、一回のGCによって生じるポーズタイムを大幅に短縮した手法である。

この手法では、objectは生成されてから通常のGCをある一定の回数以上生き残ると寿命の長いobjectとみなされ、通常のGCの対象外となるが、その回数（Advancement Threshold）をいくつに設定するかという問題が残される。

本稿では、Advancement Thresholdの数字の持つ意味の重みがアプリケーションによって変わってしまうことを指摘し、アプリケーション実行中のセルの消費状態を監視するセル消費監視プロセスの設置を提案する。セルの消費状態によって、Advancement Thresholdを動的に調整することで長寿命領域中の無駄なゴミを最小限に抑えることを考え、セルの消費状態に対してAdvancement Thresholdをどのように調整するべきかを知るために現在行なっている実験データの紹介を行ない、今後の展望について述べる。

2 Generation Scavenging GC

GCにobjectの寿命の概念を取り入れ[LH83]、objectを長寿命のものと短寿命のものに分けることによって、通常のGCを短寿命のobjectに限定して行ない、長寿命のobjectにかかるGCの時間を削減し、一回のGCにかかる時間を短くするというGCの手法がある。Generation Scavenging GC（図1）はCopying-based GC[Bak78]にobjectの寿命の概念を取り入れたものである。この方法ではaddress spaceを生成領域、2つの短寿命領域、そして長寿命領域に分けている。objectの生成は生成領域に対して行ない、生成領域が使い尽くされると、GCが始まる。生成領域、および短寿命領域中の生きているobjectは、もう一つの短寿命領域にコピーされる。短寿命領域のGCを何回か生き残ったobjectは長寿命であると判断され、長寿命領域に移される。これをTenuringという。長寿命領域に移されたobjectは通常のGCの対象外となる。

生き残ったobjectがすべて各領域に移されると、2つの短寿命領域は入れ換えられてGCは終了する。以上にあげた通常のGCは、生成領域と短寿命領域中の生きたobjectに対してのみ行なわれ、長寿命領域に移されたobjectは通常のGCの対象外となるので、一回のGCにかかる時間は大変短いものになる。また、生成領域と短寿命領域を主記憶にとり、長寿命領域を仮想記憶にとることにより、無駄なページフォルトを防ぎ多くのメモリを効率的に使用することができる。Advancement Thresholdの値によってはそのobjectがTenuringしたすぐ後に死んでしまって長寿命領域を無駄にする（Tenured Garbage[UJ88]）ことがあるので、Advancement Thresholdをいくつに設定するかという問題が残される（Tenuring Problem）。また、各objectは自分の生存時間を管理するためのage countを持っているが、そのカウント操作のコストがかかる。

3 Opportunistic GC(OGC)

Opportunistic GC(以下OGC)[WM89]において、Advancement Thresholdは1から2の間が良いとされている。それは、次のような理由による。

Advancement Thresholdを1に設定すると、生成領域の中で生きているobjectはただちに長寿命領域にTenuringされる。しかしこの場合は、領域がscavengeされる直前に生成されたobjectは死ぬまでの十分な時間を与えられずにTenuringされてしまう。その結果、長寿命領域に移された寿命の短いobjectはすぐにゴミとなってしまい、長寿命領域の使用効率が大幅

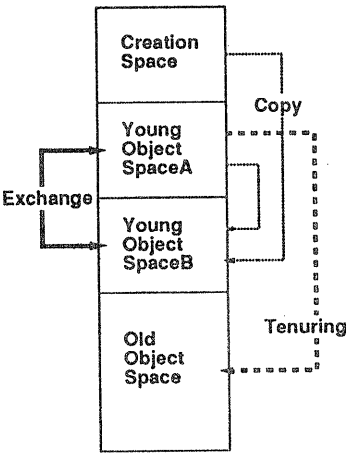

図 1: Memory Usage of Generation Scavenging GC

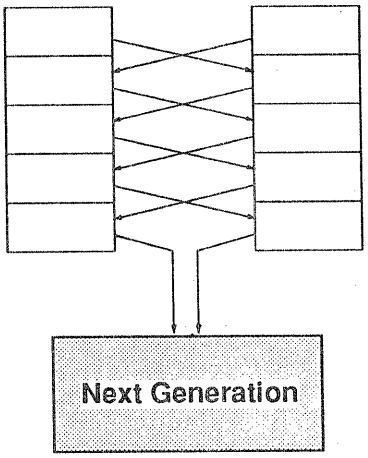

図 2: Bucket-Brigade

に下がってしまう。

Advancement Threshold を 2 に設定すると、前述のような問題は生じない。すなわち、領域が scavenge される直前に生成された object はすぐに長寿命領域に Tenuring されるわけではなく、短寿命領域中の semi space に 1 回コピーされ、次の scavenge の時に長寿命領域に Tenuring されるからである。しかし、この場合は長寿命領域に Tenuring される object は 2 度コピーされるので、コピーのコストがかかってしまう。Advancement Threshold を 2 よりも大きくする場合には、Threshold を増やすコストに比べて長寿命領域へ Tenuring される object が減る割合が少ない。従って、Advancement Threshold を 2 よりも大きくする意味はあまりない。Advancement Threshold は、1 から 2 の間が良いとされている。

OGC は、Generation Scavenging GC に **bucket-brigade** の手法を使用することにより、各 object が age count を持つことなしに AdvancementThreshold を 1 から 2 の間に設定できるようにした手法である。bucket-brigade の手法（図 2）は、一つの領域を二つに分け、各領域をさらにいくつかの bucket に分ける。コピーが始まると、各 bucket はずらされるようにしてもう一つの領域の bucket に移される。bucket の数だけずらされた object は自動的に次の世代へ移ることができる。したがって、各 object が age count を持つことなしに寿命の管理ができる。

この手法では生成領域に境界 (**watermark**) を設けることにより、生成領域中の object を長寿命と判断して良いものと、まだ長寿命と判断するには十分な時間を経過していないものとに分けることによって、Advancement Threshold を 1 から 2 の間に設定できるようにしている。この方法では境界を移動することにより、Threshold を 1 から 2 の間に自由に設定することができる。これらは各 object に age count を持たせることなしに、bucket-brigade の手法により実行される（図 3）。

4 アプリケーションによるセル消費の違い

Advancement Threshold を設定する際には、次のことを考慮に入れる必要がある。

ある一定量のセルを消費する時間がアプリケーションによって異なるので、セル消費のペースの非常に速いアプリケーションでの GC を 1 回生き残る場合とセル消費のペースの比較的遅いアプリケーションでの GC を 1 回生き残る場合とでは、object の生存時間が異なってくる。したがって、Advancement Threshold

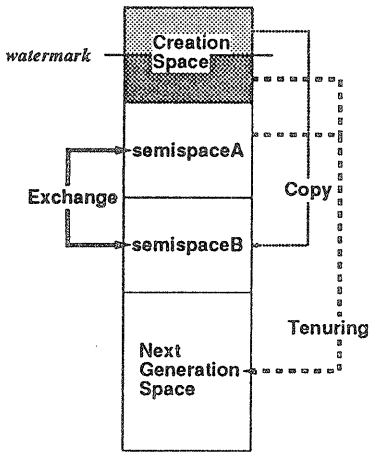

図 3: Memory Usage of OGC

が同じであるにもかかわらず、セル消費のペースが非常に速いアプリケーションでは Tenured Garbage の割合が多くなり、長寿命領域が無駄になるという問題が起こってくる。つまり、アプリケーションによって Advancement Threshold の数字の持つ意味の重みが変わってしまう。アプリケーションによって、さらにはセルの消費のしかたによって動的に Advancement Threshold を決定することによって、より効率の良い GC を行なうことができる。

5 セル消費状態監視プロセスの設置

前述のような問題を解決する方法として、セルの消費のしかたに合わせた GC を行なうことを提案する。具体的には、本稿で提案する GC を実現するために、アプリケーション実行中のセルの消費状態を監視するセル消費状態監視プロセスを設置し、リスト処理プロセスと並列に実行する。このセル消費状態監視プロセスは、セル消費の状態をリスト処理プロセスに伝える。一方リスト処理プロセスはセル消費監視プロセスから伝えられるセルの消費状態によって Advancement Threshold を動的に調整して GC を行なうことで、Tenured Garbage の割合を最小限に抑え、長寿命領域を有効に使用する。

セルの消費状態を表す項目として次のようなことがあげられる。

☆ 単位時間に消費するセルの数

☆ GC 間のインターバル

☆ Tenuring したすぐ後に死んでしまう Tenured Garbage の割合

☆ 1 回の GC で各領域に生き残るセルの割合

これらの状態を監視プロセスによって監視させ、その状態に最適であるように Advancement Threshold を動的に調整する。GC 間のインターバルが短い、あるいは Tenured Garbage の割合が大きい時には、生成領域中の境界を下げることによって長寿命領域に Tenuring する object の数を減らし、GC 間のインターバルが長い、あるいは Tenured Garbage の割合が小さい時には、生成領域中の境界をあげることによって長寿命領域に Tenuring する object の数を増やす。
セルの消費状態に対してどのように Advancement Threshold を調整するべきかを知るために、実際に様々なアプリケーションを動かして Advancement Threshold の値を変えながら、監視プロセスの監視すべきセルの状態項目のデータをとる。そして、各セルの状態に最適な Advancement Threshold の値を見つけ出し、セル消費状態監視プロセスに適応させることを考える。

6 実験

セルの消費状態に最適な Advancement Threshold の値を知るために、セル消費のペースが違う様々なアプリケーションを実行させてデータをとることを考える。
実際にセル消費監視プロセスを設置する時は、Lisp などのリスト処理のインタプリタに組み込み、リスト処理プロセスと並列に実行させるが、ここで 1 つ注意せねばならないことがある。各アプリケーションが陽にセルを消費する他にインタプリタ自身が陰にセルを消費することを考慮に入れると、インタプリタによって

同じアプリケーションを動かしてもセルの消費状態は大きく異なることが考えられる。したがってインタプリタを通さずに純粋にセルを消費し続けるようなアプリケーションを動かした。GC はOGC のアルゴリズムをを採用して実験を行なっている。
実験はS-4/490 で行なった。

6.1 space size

各領域のサイズは使用した機種のページサイズを考慮して、生成領域と2つの短寿命領域にそれぞれ約160K、長寿命領域には約960K 確保した。Lisp などのインタプリタでアプリケーションを実行させる場合には GC 時の生成領域からの object の生存率を考えると短寿命領域は生成領域よりも少なくて十分なはずであるが[Ung84]、この実験においては生成領域中のすべての object が生き残ることもあり得るので各短寿命領域は生成領域と同じサイズだけ確保した。

6.2 アプリケーション

セルを消費し続けるようなアプリケーションとして、リストがリニアに伸び縮みするようなもの、リストが単純な tree 型を形成するもの2通りについて実験を行なった。両者ともに与えられた回数だけ cons と cut を繰り返すもので、何回かの GC の後、長寿命領域がいっぱいになった時点で実行をやめる。または、GC が100 回を越すとその時点で実行をやめるようにした。
GC において、生成領域を10 等分することによってAdvanthment Threshold を0.1 刻みで1.0 から2.0 の間に設定してデータをとった。1回の GC が終ると、各領域から生き残ったセルの数を数えて各領域のセルの最大個数に対する割合を求め、最終的に1回の GC で生き残ったセルの割合の平均をとり、各領域からのセルの生存率とした。1回の実行では長寿命領域を除く各領域でのセルの生存率はGC が複数回起こってもほとんど変わらないので、平均をとることで十分であると考えられる。長寿命領域での生存率はGC の起こる度に変わるので、上記のように割合の平均を求めて生存率を出す他に、実行の最後に長寿命領域中で生き残っていたセルの割合についても計算してその割合を出した。

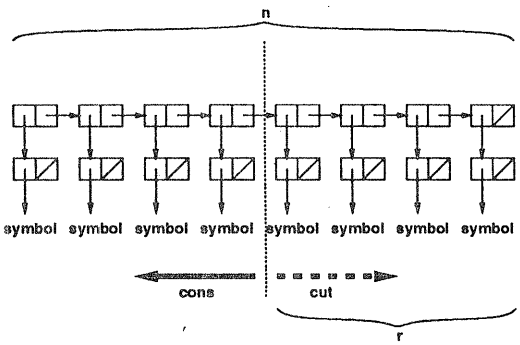

図4: リニア型アプリケーション

6.2.1 リニア型アプリケーション

セルがリニアに伸び縮みするようなアプリケーションとして図4のようにcons をn 回と cut をr 回することを単純に繰り返すようなものをリニア型アプリケーションとする。cons と cut の数の組合せ4パターンの実験を行なった（表1）。

アプリケーション名	cons(n の数)	cut(r の数)
AlistA	100	50
AlistB	10000	9500
AlistC	30000	25000
AlistD	10000	5000

表1：リニア型アプリケーションの実行

図5、図6は、最終的な長寿命領域中のセルの生存率と、各GC の後の長寿命領域中のセルの平均個数による生存率を各 Advancement Threshold に対応させたグラフである。

6.2.2 tree 型アプリケーション

セルが単純に tree 型を形成するアプリケーションとして図7のように tree をn 段作ってはr 段 cut することを繰り返すようなものを tree 型アプリケーショ

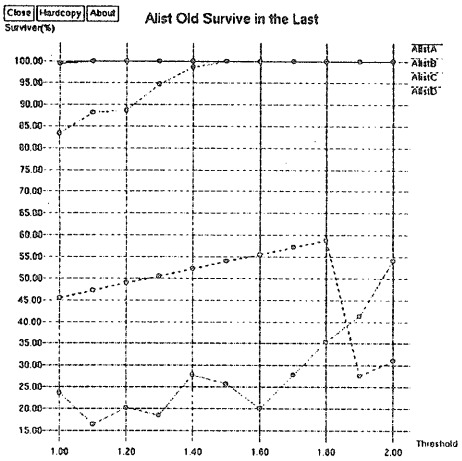

図 5: Alist Old Surviver in the Last

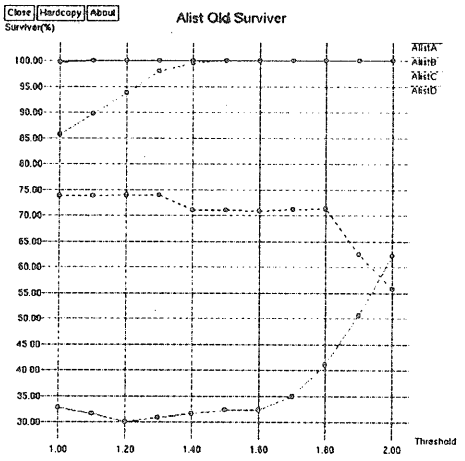

図 6: Alist Old Surviver

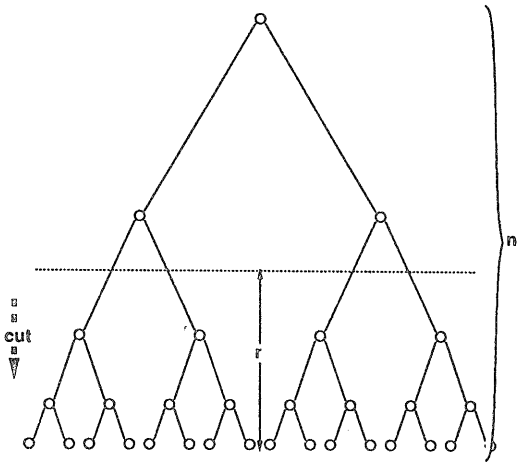

図 7: tree 型アプリケーション

ンとする。tree の段数と cut の数の組合せ 3 パターンの実験を行なった（表 2）。

アプリケーション名	tree(n の数)	cut(r の数)
TreeA	10	5
TreeB	15	10
TreeC	15	3

表 2：アプリケーション 2 の実行

図 8、図 9 は、最終的な長寿命領域中のセルの生存率と、各 GC の後の長寿命領域中のセルの平均個数による生存率を各 Advancement Threshold に対応させたグラフである。

7 考察

図を見るとわかるように、基本的には Advacement Threshold の大きい方が最終的な長寿命領域生存率は高い。しかし、アプリケーションによっては図 6 の AlistC のようにそうでない場合もある。図 5、図 6、図 9 に共通して見られることは、Advancement Threshold が大きくなるにつれ、一定の間隔で生存率が上がったり下がったりしていることである。セルの消費の仕方によってその値は変わるが、これは Advancement Threshold とアプリケーションと長寿命領域の大きさ

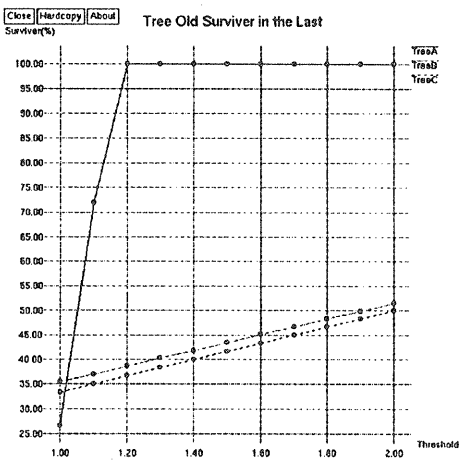

図 8: Tree Old Surviver in the Last

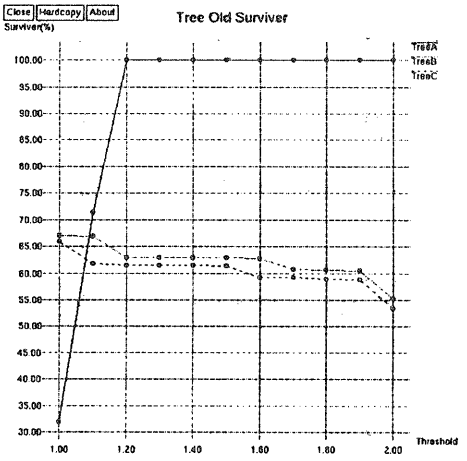

図 9: Tree Old Surviver

の兼ね合いであるところで極端に下がるようなことが起こると考えられる。さらに数多くのアプリケーションを実行してデータをとることによって、その傾向も明らかにされると思われる。

参考文献

[Bak78] Henry G. Baker, Jr. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[LH83] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[UJ88] David Ungar and Frank Jackson. Tenuring Policies for Generation-Based Storage Reclamation. *OOPSLA '88 Proceedings*, pages 1–17, September 1988.

[Ung84] David Ungar. Generation Scavenging:A Non-disruptve High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, May 1984.

[WM89] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. *OOPSLA '89 Proceedings*, pages 23–35, October 1989.