効率のモデルに基づきヒープサイズを自動調節する世代 GC 方式

吉川隆英[†] 近山隆[††]

世代 GC 方式は，データ割付領域を生成後間もないデータを配置する新世代領域と長寿命データを配置する旧世代領域とに分割することにより，長寿命データが何度も GC を経験するのを防ぐとともに，データ局所性の向上を図るメモリ管理方式である．世代 GC 方式においては，新世代領域から旧世代領域への移動（殿堂入り）時期の適切な選択が性能を大きく左右する．通常，殿堂入り時期はデータの GC 経験回数によって決定される．そして，GC は新世代領域を使いきった時点で発生する．したがって，新世代領域サイズを動的に変更すれば，殿堂入り時期は動的に変更できる．本稿では，GC 時に回収されるゴミの比率のモデルに基づきデータの平均余命を推定，その結果に沿って新世代領域サイズを動的に変更することによって，殿堂入り時期を適切に調節する世代 GC 方式を提案する．また，この方式を並行並列論理型言語処理系 KLIC に実装し評価を行った結果も述べる．

A Generational GC Scheme that Dynamically Adjusts New Generation Heap Sizes Based on an Efficiency Model

Takahide Yoshikawa[†] and Takashi Chikayama[††]

A generational garbage collection segregates heap objects into multiple areas by their age, and garbage-collects areas containing older objects less often than those of younger ones. In a version of this scheme using two areas, new heap objects are allocated to the younger generation area, and advanced to the older generation area after a while. With an appropriate advancement policy, it can avoid repeated inspections of long-lived objects and improve reference locality. When to advance objects to the older generation is usually decided by the number of GCs that the object experienced. A GC occurs when the younger generation area is filled up. So the advancement policy can be dynamically modified by adjusting the new generation area size. In this paper, we propose an adjustment scheme based on *garbage ratios*, by which one can estimate life expectancy of short-lived objects. And we also report the evaluation results of this sheme when applied to an implementation of a concurrent and parallel logic programming language, KLIC.

1. はじめに

一般に，自動メモリ管理を行う言語においては，ガーベジコレクション（GC）がプログラム実行時間全体のうちの無視できない時間を占める．そのため GC の性能は処理系の性能を大きく左右する．GC の時間を短縮するため，これまでに様々なアルゴリズムが提案されてきた[1]．世代 GC 方式はこの 1 つである．世代 GC 方式は「多くのデータは生成後すぐに無効になる．しかし一部のデータは非常に長い時間生き続ける」という性質[2]に基づき，データ割付領域を生成後間もないデータを配置する新世代領域と長寿命データを配置する旧世代領域とに分ける方式である．

† 東京大学工学系研究科
School of Engineering, the University of Tokyo
†† 東京大学新領域創成科学研究科
School of Frontier Sciences, the University of Tokyo

世代 GC 方式において，新たに生成されたデータは，何回かの GC を経た後，旧世代領域に移される．これを「殿堂入り」という．この殿堂入りの時期の判定が適切であれば，長寿命データが何度も GC を経験することを避けることができ，GC 処理の手間を削減できる．また，生成されて間もないデータを新世代領域にまとめられるため，データの局所性が向上する．殿堂入りの適切な時期は，プログラムによって，また 1 つのプログラムの中でもその実行フェーズによって異なる．最適な殿堂入り時期をどのように決定するかという問題は「殿堂入り問題（Tenuring Problem）」と呼ばれ，いくつかの解決手法がこれまでにも提案されている[3)~7)]．

本稿では，新世代領域のサイズを，データの寿命や GC 効率のモデルに基づいて動的に変更し，殿堂入り時期を適切に調節することによって，旧世代領域への過度の殿堂入りを防ぎ，新世代領域でのデータ局所性

を向上させるメモリ管理方式を提案する．そして，この方式を並行並列論理型言語処理系 KLIC[9]に実装し，評価を行う．

2. 世代 GC 方式

2.1 世代 GC 方式

世代 GC 方式を採用している処理系では，「多くのデータは生成後すぐに無効になる．しかし一部のデータは非常に長い時間生き続ける」という性質[2]に基づき，処理系がデータを割り付けるメモリ領域（ヒープ）を，新世代領域と旧世代領域とに分割する．そして，使われなくなったデータ（ゴミ）が発生しやすい新世代領域に対する GC を頻繁に行い，ゴミが発生しにくい旧世代領域に対する GC をほとんど行わないようにする．こうすることによって，

- 新世代領域では，GC が頻繁に走ることにより，メモリ利用効率と局所性が向上，
- 長寿命データを分離し，それらに対する GC 頻度を下げることにより，長寿命データが何度も GC にかかることを防止，

という効果を得ることが期待できる．

2.2 殿堂入り問題

世代 GC 方式において，データが旧世代領域に移されることを「殿堂入り」という．そして，この殿堂入り時期をどのように設定するかは GC の効率に大きな影響を与える．

殿堂入り時期の適切な値は，プログラムによって，また 1 つのプログラムの中でもその実行フェーズによって異なる．適切な殿堂入り時期をどのように決定するかという問題は「殿堂入り問題（Tenuring Problem）」と呼ばれ，いくつかの解決手法がこれまでにも提案されている[3)~7)]．

2.3 今回評価に用いる世代 GC 方式

世代 GC 方式には様々な実現方式があるが，本稿では，新世代 2 面，旧世代 2 面で，

- 新しいデータは新世代領域に割り付けられる，
- 新世代領域が溢れると新世代 GC が発生．データは新世代コピー先領域にコピーされる，
- 新世代領域で 2 回目の GC を経験すると旧世代領域にコピーされる（殿堂入り），
- 旧世代領域の空き領域サイズが新世代領域のサイズより小さくなると旧世代 GC が発生．データは旧世代コピー先領域にコピーされる，

という方式を評価に用いる（**図 1**）．

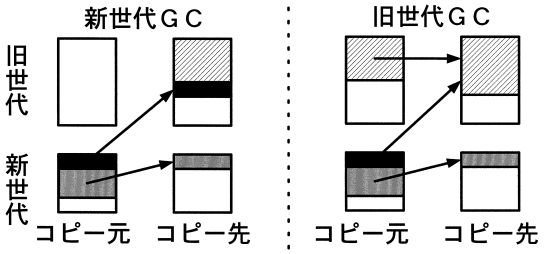

図 1 今回評価に用いる世代 GC 方式
Fig. 1 Generational GC.

3. 従来手法と提案手法の概略

3.1 データの寿命

プログラム実行中にメモリ上に割り付けられたデータは，処理が進むにつれ，やがて使われなくなる．GC は，この使われなくなったデータ領域を再利用可能にする処理である．そして，データの寿命とは，データが生成されてから使われなくなるまでの期間の長さである．データの寿命は以下のように定義される．

データの寿命は，そのデータが有効である間に割り付けられたメモリ量で表される．

すると，ある程度のメモリ割付けごとに GC が発生する場合，何回も GC を経験したデータは長寿命ということになり，GC をほとんど経験しないうちに使われなくなったデータは短寿命ということになる．

3.2 従 来 手 法

これまでの殿堂入り問題解決手法には主に以下の 2 つがある．1 つ目は Ungar らの提案した Demographic Feedback-Mediated Tenuring（DFMT）[3),4)]である．これは，新世代領域中の殿堂入り候補が，ある一定量を超えるまで殿堂入りさせず，超えたときに古いデータから順に殿堂入りさせる手法である．

2 つ目は Wilson らが提案した Opportunistic Garbage Collector（OGC）[5)]と，田中らが提案した Adaptive Garbage Collection（AGC）[6),7)]である．これは，メモリ領域を「生成領域」「短寿命領域」「長寿命領域」の 3 つに分け，生成領域に “Water Mark” を設け，GC 時に，

- Water Mark 以上（Water Mark 以前に生成）のデータと短寿命領域データを長寿命領域にコピー，
- Water Mark 以下のデータは短寿命領域にコピー，

する手法である．これにより，殿堂入りまでの GC 経験回数（Advancement Threshold: AT）を 1〜2 の間で自由に設定できる．

3.3 従来手法の問題点と改善の方針

DFMT では，Sliding Compaction 方式を採用するか，GC 経験回数を記録するなどして，データがどのくらい生きてきたかをつねに正確に把握しておかなければならず，メモリ管理が複雑になってしまう．一方，OGC や AGC では，AT を 1〜2 の間で自由に設定できるが，それ以上の値に設定するのが非常に困難であり，また一定の割付け量ごとに GC を起こすため，プログラムの実行フェーズの大きな変化には対応しきれない．

さらに，これまでの殿堂入り問題解決手法はすべて，旧世代領域のゴミ（Tenured Garbage: TG）を減らすことや，旧世代 GC の回数を減らすことを主眼に置いてきた．しかし，メモリの階層化と 1 次キャッシュ容量の増加により，多少 TG が増えたり，旧世代 GC 回数が増えたりしても，新世代領域が 1 次キャッシュに収まるようにした方が高速になる場合もある．

そこで，本稿では，処理系の実装を簡単にするため，AT の値は 2 に固定し，以下のような方針に従う世代 GC 手法を提案する．

- 新世代領域サイズを変更することによって，GC 頻度を調節する．これにより，殿堂入りのタイミングを広い範囲で自由に設定できるようにする．
- 新世代領域サイズを，過度の殿堂入りが発生しない程度に小さく抑え，新世代領域の局所性を改善させ，キャッシュヒット率の向上を図る．

4. 殿堂入り時期の調節

本章では，まず，新世代領域サイズを調節することによって殿堂入り時期の調節を行う方法を述べる．次に，新世代領域サイズを調節する方針を決めるためのモデルとしてデータ余命のモデルとゴミ比率のモデルを導入する．

4.1 殿堂入り時期調節の方法

今回評価に用いる世代 GC 方式では，2 回目の GC を経験した時点で，そのデータを殿堂入りさせることにしている．これは，殿堂入り時期を GC3 回以上経験時とすると，GC 方式によっては，データに GC 経験回数のタグをつけるなどして，生存時間を管理することが必要なのに対し，殿堂入り時期を GC2 回目経験時とすると，GC 終了時に，その時点でのメモリ割付点を記録しておくだけで殿堂入り候補を識別できるからである．

殿堂入りまでの GC 経験回数を固定したまま殿堂入り時期を調節するには GC の頻度を調節する必要がある．GC は新世代領域を使い切った時点で発生するので，GC 頻度は新世代領域サイズを変更してやれば調節することができる．すなわち，新世代領域サイズを適切に設定すれば殿堂入り時期は適切に設定することができる．まとめると，

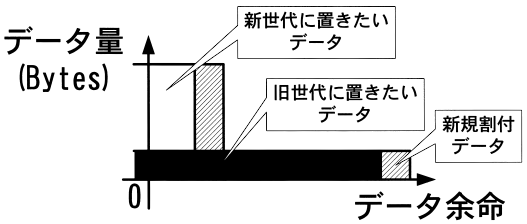

図 2 データ余命の分布
Fig. 2 Data Life Model.

新世代領域サイズを変更してやれば GC の頻度を調節できる．殿堂入りまでの GC 経験回数を固定していても，GC 頻度を調節できれば殿堂入り時期を調節することができる．

ということである．

4.2 データ余命のモデル

ある時点において，データがあとどのくらいの GC を経験するかを示す値をデータの余命とする．世代 GC 方式においては，余命の短いデータが旧世代領域に入ると，以下のような問題を生ずる．

- 旧世代 GC が頻繁に発生する．
- 旧世代領域のメモリ利用効率が低下することになり，世代 GC の利点が失われる．

したがって，世代 GC 方式における適切な殿堂入りとは，余命の短いデータを旧世代領域に持ち込まないということになる．

いま,「メモリ上のデータは，多くは生成後すぐに無効になる，しかし一部は非常に長い時間生き続ける」とするならば，ある時点でのデータの余命の分布は**図 2** のようにモデル化できる．

そして，このモデルにおいて，適切な殿堂入りとは，図 2 の余命の短いデータ（図の白い部分）を新世代領域に配置し，余命の長いデータ（黒色の部分）を旧世代領域に配置するということである．

4.3 ゴ ミ 比 率

今回評価に用いる世代 GC 方式において，新世代 GC にかかる手間は，GC 時にコピーしなくてはならないデータの総量，すなわち，GC 時点での新世代領域中の有効なデータの総量に比例する．

そこで，GC の手間を測る尺度としてゴミ比率を定義する．ゴミ比率は，新世代 GC 発生時点で新世代領域にあった全データのうち，どのくらいのデータがゴ

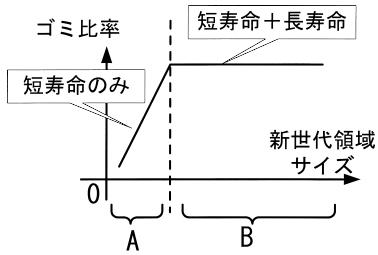

図 3 ゴミ比率のモデル
Fig. 3 Garbage ratio model.

ミとして回収されたかを示すもので，以下のようにして定義される．

$$\frac{n_f - n_t + o_f - o_t}{n_f} \quad (1)$$

n_f ：新世代 GC 前の新世代データ量
n_t ：新世代 GC 後の新世代データ量
o_f ：新世代 GC 前の旧世代データ量
o_t ：新世代 GC 後の旧世代データ量

すると，このゴミ比率が高くなればなるほど，新世代 GC 時にコピーする（生きている）データの量が減るため，1 回の新世代 GC にかかる手間は小さくなるということになる．

4.4 ゴミ比率のモデル

データ余命が 4.2 節のモデルに従うとすると，ゴミ比率と新世代領域サイズとの関係は**図 3** のようになると考えられる．

これは，

A) 新世代領域サイズが小さすぎるとき
　短寿命データのみが新世代 GC で回収されている状態．新世代領域サイズを拡大すると，より多くの短寿命データが新世代 GC で回収され，ゴミ比率が向上する．
B) 新世代領域サイズが十分大きいとき
　長寿命データの一部が新世代 GC で回収されている状態．新世代領域サイズを拡大しても，長寿命データの回収はあまり見込めないため，ゴミ比率はほとんど改善されない．

ということを示している．

4.5 ゴミ比率に基づく新世代領域サイズ調節

世代 GC は，短寿命データを新世代領域に配置し，長寿命データを旧世代領域に配置する GC 方式である．この効果を最大限に引き出すためには，

- 短寿命データを旧世代領域に入れない，
- 長寿命データは，なるべく早く旧世代領域に追い出すことによって，新世代領域の局所性を向上させる，

ことが必要であると考えられる．すると，4.4 節のゴミ比率のモデルに基づいて適切な殿堂入りを行うためには，

A) 新世代が小さすぎるとき（図 3 の A の範囲）
　より多くの短寿命データを新世代 GC で回収するために新世代領域サイズを拡大する．
B) 新世代が大きすぎるとき（図 3 の B の範囲）
　新世代領域サイズを多少変化させてもゴミ比率はほとんど変わらない．つまり，新世代 GC の手間はあまり変化しない．そこで，より多くの長寿命データを旧世代領域に追い出すことによって新世代領域の局所性を向上させるため，新世代領域サイズを縮小する．

とすればよい．つまり，適切な殿堂入りを行うためには，新世代領域サイズを図 3 の AB の境界の値に設定してやればよい．AB の境はプログラムによって，また，プログラムの実行フェーズによって変動するため新世代領域サイズを固定していると，不適切な殿堂入りが発生してしまう．そこで，適切な殿堂入りを行い続けるためには，ゴミ比率の値の変動を見ながら適当に新世代領域サイズを調節しなくてはならない．

4.6 実例との比較

4.4 節で示したゴミ比率のモデルを検証するために，世代 GC 方式を採用している実際の処理系で新世代 GC のゴミ比率と新世代領域サイズの関係の測定を行った．測定に用いた処理系は並行並列論理型言語 KL1 の処理系 KLIC3.003 版[9]である．測定に用いたテストプログラムは KLIC3.003 版付属のテストプログラムのうち，

- life.kl1：ライフゲーム
- hanoi.kl1：ハノイの塔

の 2 つである（詳細は付録を参照）．そして，各プログラムに対して，**「新世代領域サイズを固定して実行し，新世代 GC のゴミ比率を毎回すべて記録する」**という作業を，様々な新世代領域サイズに対して行った．その結果を**図 4**，**図 5** に示す（グラフにはゴミ比率の平均値と標準偏差が示されている）．

図 4 から，life.kl1 では新世代領域サイズを拡大していくと 256 KB 付近まではゴミ比率が大きく向上するが，512 KB 付近で 100%近くになって飽和してしまう．

図 5 から，hanoi.kl1 では，新世代領域サイズが 16 KB の時点で，すでにゴミ比率はほぼ 61〜62%の値に飽和している．この状態で新世代領域サイズを拡大していくと，ゴミ比率の分布範囲が狭くなるだけで，平均値にはほとんど変化がない．

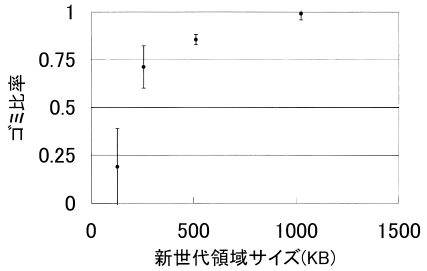

図 4 life.kl1 のゴミ比率
Fig. 4 Garbage ratio (life.kl1).

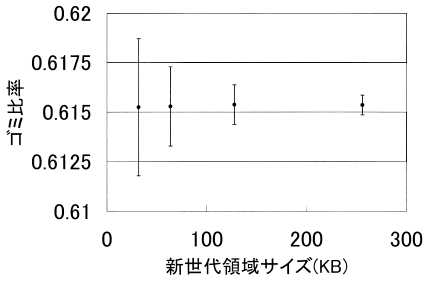

図 5 hanoi.kl1 のゴミ比率
Fig. 5 Garbage ratio (hanoi.kl1).

よって，ゴミ比率と新世代領域サイズとの関係には，

- 新世代領域サイズを大きくしていくと向上する，
- しかし，ある値を境に飽和する，
- 飽和して落ち着く値はプログラム（あるいは，その実行フェーズ）によって異なる，

という性質があるといえる．そして，この性質は 4.4 節で示したゴミ比率のモデルと一致する．

5. キャッシュの影響

本方式において，新世代領域サイズを縮小する目的は，新世代領域の局所性を向上させ，キャッシュヒット率を向上させることである．そこで本章では，キャッシュヒット率を向上させるために最適な新世代領域サイズを求める．

5.1 キャッシュと実行速度の関係

新世代領域サイズとキャッシュサイズとの関係が性能に影響を与えるのは，データの主要部分がキャッシュに入りきるようなプログラムの場合である．このようなプログラムでは新世代領域サイズを短寿命データが入りきる範囲で縮小してもゴミ比率はほとんど変わらない．そこで，このような性質を持つ KLIC 付属テストプログラム，

- mastermind.kl1：数当てゲーム
- kkqueen.kl1：N クイーン

において，新世代領域サイズをある値に固定するこ

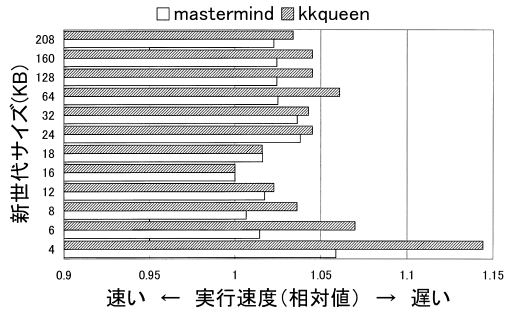

図 6 新世代領域サイズと実行速度（K6-2）
Fig. 6 New generation size vs. execution speed (K6-2).

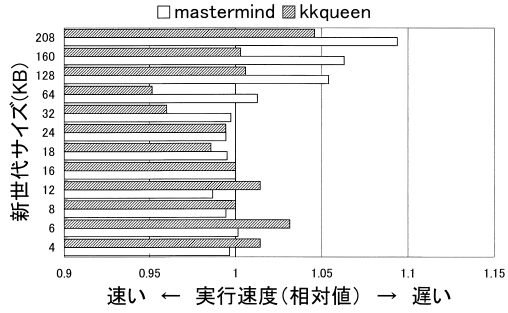

図 7 新世代領域サイズと実行速度（K6-3）
Fig. 7 New generation size vs. execution speed (K6-3).

とによって，新世代領域サイズと実行速度の関係を測定した．その結果を**図 6**，**図 7** に示す．なお，実験環境は図 6 の側が AMD K6-2（1 次データキャッシュ 32 KB），2 次データキャッシュ 1 MB，メモリ 128 MB，OS は Solaris 2.6，図 7 の側が AMD K6-3（1 次データキャッシュ 32 KB，2 次データキャッシュ 256 KB），3 次キャッシュ 512 KB，メモリ 128 MB，OS は Linux 2.2.9 である．

5.2 最適な新世代領域サイズ

図 6 の K6-2 での実験では新世代領域サイズを CPU の 1 次データキャッシュサイズの半分の値にしたときが最も実行速度が速い．これは，プログラム実行時，最も局所性が悪いメモリアクセスが発生するのが GC のときであり[8]，GC 時に新世代領域 2 面分が CPU の 1 次データキャッシュに載るようにすれば，そのときが最も実行効率が良いためである．

そして，新世代領域サイズを 1 次データキャッシュの半分以下にすると，キャッシュヒット率はほとんど変化せず，新世代・旧世代 GC 回数が増えるため，実行速度は低下する．逆に，キャッシュサイズの半分以上にすると新世代・旧世代 GC 回数は減少するが，キャッシュヒット率が低下するため，実行速度は低下してしまう．

そこで，新世代領域サイズの初期値と最小値を，プ

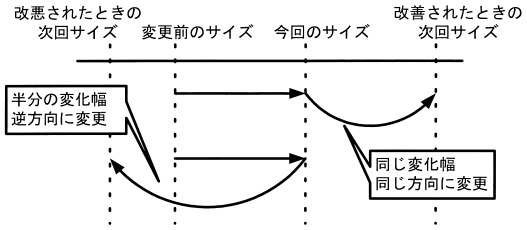

図 8 サイズ調節の方針
Fig. 8 Memory size adjustment policy.

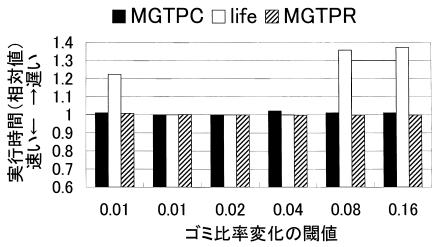

図 9 サイズ調節の閾値
Fig. 9 Memory size adjustment threshold.

ログラムを実行するプロセッサの 1 次データキャッシュサイズの 1/2 とすることとした．

なお，図 7 の K6-3 での実験では 2 次キャッシュまでがプロセッサと同クロックで動作するため，問題によっては，1 次データキャッシュ（32 KB）の影響よりも 2 次データキャッシュ（256 KB）の影響の方が大きく出ている．

6. 実装と評価

ゴミ比率のモデルに基づき新世代領域サイズを自動調節する世代 GC 方式を KLIC-3.003 に実装した，本章ではその実装方法と実験結果を示す．

6.1 新世代領域サイズ調節の条件

6.1.1 サイズ調節の方針

4.5 節で，新世代領域サイズが小さすぎるとき（図 3 の A の範囲）に拡大，大きすぎるとき（B の範囲）に縮小することによって新世代領域サイズを調節することとした．しかし，AB の境界はプログラムによって，またプログラムの実行フェーズによって変化するため，A の範囲にあるのか B の範囲にあるのかを，実行中に動的に判定し直す必要がある．

そこで，プログラム実行中，つねに GC 時のゴミ比率と新世代領域サイズを記憶しておき，新世代領域サイズを変更した結果，

- ゴミ比率の改善が閾値を上回れば，同じ変化幅で同じ方向にさらに変更，
- ゴミ比率の改悪が閾値以上ならば，変更前のサイズから，前回の半分の変化幅で逆方向に変更，

とすることとした（**図 8**）．なお，変化幅の縮小はサイズの振動を最適値に向けて収束させるために行っている．

この具体的な手順は，新世代 GC 10 回につき 1 回，新世代領域サイズを 2 倍に拡大，50 回につき 1 回，半分に縮小し，その後の GC ごとに，ゴミ比率の変化に応じて以下の操作を行うこととした．

- 拡大してゴミ比率が閾値以上改善：
  同じ変化幅でさらにサイズを拡大
  （次サイズ=現サイズ+今回の変化幅）
- 拡大してゴミ比率が閾値以上改悪：
  変化幅を半分にし前回のサイズから縮小
  （次サイズ=前サイズ−今回の変化幅/2）
- 縮小してゴミ比率が閾値以上改善：
  同じ変化幅でさらにサイズを縮小
  （次サイズ=現サイズ−今回の変化幅）
- 縮小してゴミ比率が閾値以上改悪：
  変化幅を半分にし前回のサイズから拡大
  （次サイズ=前サイズ+今回の変化幅/2）
- 閾値以下の変化：
  前回サイズに戻し，サイズ調節を終了
  （次サイズ=前サイズ）

6.1.2 サイズ調節の閾値

次にサイズ調節の閾値を考える．サイズ調節の閾値を適当な値に固定したときの実行速度とサイズ調節の閾値のグラフを**図 9** に示す．なお，実験環境は AMD K6-2（1 次データキャッシュ32 KB），2 次データキャッシュ1 MB，メモリ 128 MB，OS は Solaris 2.6，測定に用いたテストプログラムは，サイズ調節がある程度発生する以下の 3 つである（hanoi や kkqueen などはプログラムの実行がほぼ均質で，最適値の変動があまりないため，この実験には用いない）．

- MGTP の実行（MGTPR）
  KL1 で書かれたモデル生成型定理証明器 MGTP naive 版[10]に TPTP ライブラリ[11]の SYN006-1_naive という問題を解かせる．
- MGTP の問題コンパイル（MGTPC）
  TPTP ライブラリの SYN036-1_naive.kl1 という問題のコンパイルを行う．
- ライフゲーム（life）
  KLIC ランタイム付属テストプログラム life.kl1 の実行を行う．

図 9 から分かるように，閾値を小さくしすぎると，新世代領域サイズが過度に変動してしまい，実行速度が低下してしまう．逆に閾値を高くしすぎると適切な

サイズに変更される前にサイズ調節を止めてしまうため，殿堂入り時期が適切な値に設定されず，実行速度が低下してしまう．そこで，サイズ調節を行うときのゴミ比率変化の閾値として，2%を採用することにした．

6.2 評　　価

ゴミ比率のモデルに基づき新世代ヒープサイズを自動調節する世代 GC を並行並列論理型言語処理系 KLIC3.003 版に実装し，その効果を測定した．評価は，テストプログラムの実行時間を，

- オリジナルの KLIC（ORG）
- 世代 GC を導入した KLIC（GGC）
- 新世代領域サイズを調節する KLIC（ADJ）

の 3 方式で比較することによって行った．オリジナルの KLIC は 2 面の Stop-and-Copy GC を採用している．評価に用いたテストプログラムは，KLIC ランタイム付属のテストプログラムである，

- life：ライフゲーム
  オリジナルでの実行時間約 0.75 sec
- hanoi：ハノイの塔
  オリジナルでの実行時間約 0.32 sec
- kkqueen：N クイーン
  オリジナルでの実行時間約 7.11 sec

と，

- MGTPC：SYN036-1_naive.kl1 のコンパイル
  オリジナルでの実行時間約 3.69 sec
- MGTPR：naive 版 MGTP で SYN006-1 を証明
  オリジナルでの実行時間約 19.9 sec

の 5 つを採用した．そして，初期新世代領域サイズは 16 KB とし，初期旧世代領域サイズは 72 KB とした（オリジナル KLIC の初期ヒープサイズは 96 KB）．実験環境は AMD K6-3（1 次データキャッシュ32 KB，2 次データキャッシュ256 KB），3 次キャッシュ512 KB，メモリ 128 MB，OS は Linux 2.2.9 である．これらの実行時間の相対値を示したグラフが**図 10** である．

図 10 の ORG が示す系列はオリジナルの KLIC での実行時間，あとの系列はオリジナルの実行時間を 1 としたときの，各手法での実行時間の相対値であり，値が小さいほど速いということを示している．なお，各系列の斜線と黒の部分は，それぞれ，実行時間内で新世代 GC に要した時間と旧世代 GC に要した時間である．そして，各手法における GC 回数は**表 1** に示す．

表 1 の GGC，ADJ の括弧内の数字は旧世代 GC の回数である．GGC で非常に新世代 GC 回数が多いのは，実行に支障が出ないかぎり新世代領域サイズの拡大を行わないようにしたためである．また，GGC の方で旧世代 GC 回数が少ないのは，積極的に旧世代領域サイズを拡大することによって旧世代 GC 回数を抑えているためである．

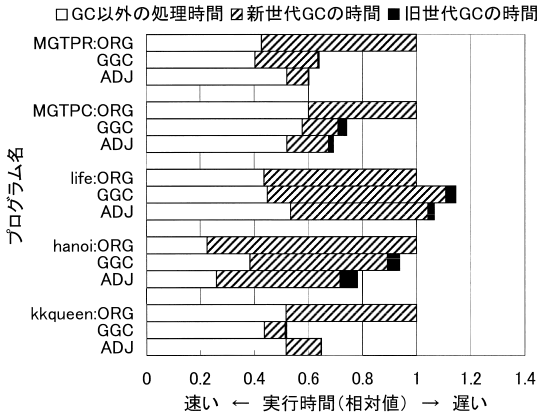

図 10　実行時間の比較
Fig. 10　Comparing execution time.

表 1　各テストプログラムでの GC 回数
Table 1　Number of GC.

	MGTPR	MGTPC	life	hanoi	kkqueen
ORG	1057	144	66	15	455
GGC	11062(4)	675(3)	60(2)	72(2)	2267(2)
ADJ	1336(6)	142(5)	58(5)	54(6)	384(4)

次に図 10 の結果について検討を行う．まず，ORG と他の 2 方式について比較を行う．life では GGC，ADJ が ORG よりも遅い．GGC が遅いのは，適切な殿堂入りができなかったためである．life には，ある程度生きてから無効になる中寿命のデータが多いため，殿堂入り時期の調節を行わない通常の世代 GC 方式では過度の殿堂入りが発生し，旧世代領域の利用効率が低下するとともに，旧世代 GC が発生しやすくなり，実行速度が低下する．一方，ADJ が遅いのは，GC 回数が少なすぎ，最適な新世代領域サイズ（512 KB 程度）にまで拡大するのに時間がかかってしまったためである．

次に GGC と ADJ についての比較を行う．

MGTPR については ADJ が GGC よりも速い．これは，MGTPR の実行中にゴミ比率が激しく変動するためである．ゴミ比率が激しく変動するため，通常の世代 GC では，実行フェーズによっては適切な殿堂入りが行えない．一方，ADJ では新世代領域サイズが実行フェーズによって適切な値に設定され，適切な殿堂入りが行われている．hanoi，MGTPC については ADJ が GGC よりも速い．これは GC 回数が少ないにもかかわらず，最適な新世代領域サイズ（384 KB 程

度）への拡大が間に合ったためである．kkqueen については，ADJ は GGC よりも遅い．これは，kkqueen が実行フェーズを 2 つ持ち，後半がゴミ比率をある程度落として（旧世代領域のゴミを増やして）でも，新世代領域サイズを小さく抑える方が有利な性質を持つためである．後半で新世代領域サイズを小さく抑える方が有利なため，拡大を積極的に行わない GGC 方式の方が速い．ADJ 方式では，実行開始直後のフェーズで積極的に新世代領域サイズを拡大したが，その後の実行フェーズでゴミ比率に大きな変化が起こらなかったため，サイズの変更が起こりにくくなり，最適値（16 KB）まで新世代領域サイズを縮小することができなかった．

以上のように，ADJ は，実行フェーズの変化によって，急激な縮小を必要とするプログラムを除けば，殿堂入り時期の調節を行わない GGC よりも高い性能を得られることが分かった．また，今回性能低下をみたプログラムについても，新世代領域サイズ縮小のアルゴリズムを改良すれば対処できると考えられる．

7. ま と め

世代 GC 方式において，新世代領域の局所性を保ちつつ，旧世代領域への殿堂入りを適切な時期に行うようにするために，新世代領域サイズをデータの寿命やゴミ比率のモデルに基づいて動的に変更する手法を提案し，それを並行並列論理型言語 KL1 の処理系 KLIC に実装，評価を行った．

その結果，実行中にゴミ比率が大きく変動するプログラムや，ある程度の GC 回数を経るプログラムにおいては，新世代領域サイズが動的に適切な値に調節されるため，適切な殿堂入り時期の設定と新世代領域のデータ局所性の確保が行われ，プログラムの実行速度が通常の世代 GC 方式よりも向上した．しかし，GC 回数が少ないプログラムでは，拡大，縮小が間に合わず，最適値に達する前に実行が終わってしまうことがあった．また，今回採用した新世代領域サイズ変更ルールは，縮小による局所性向上よりも拡大による旧世代領域ゴミの削減を重視したため，ゴミ比率の変化が少ないプログラムでは，サイズの縮小が積極的に行われず，最適値までの縮小が間に合わない場合があった．

本手法は，新世代領域サイズを変更するという単純な方法によって，殿堂入り時期の適切な調節が行っているため，今回評価に用いた各世代 2 面の Stop-and-Copy 方式世代 GC 以外にも，多くの世代 GC 方式に適用可能であると考えられる．

今後の課題としては，以下のようなことがあげられる．

- 閾値と拡大縮小ルールの変動
  今回，サイズ調節に必要な閾値を 2%に固定し，かつ，縮小のルールを消極的にした．しかし，閾値や縮小の消極策は，変更したサイズの大きさや，実行フェーズに応じて変えていく必要があると考えられる．
- より長い履歴の参照
  今回の実装では，単純に 1 回前のゴミ比率と新世代領域サイズを見て新世代領域サイズの変更を行っている．この履歴をもっと長くとれば，過大な振動を起こさずに，より大きなサイズ調節が可能になると考えられる．

参 考 文 献

1) Wilson, P.: Uniprocessor Garbage Collection Techniques, *IWMM'92* (1992).
2) Lieberman, H. and Hewitt, C.: A real-time garbage collector based on the lifetimes of objects, *Comm. ACM*, Vol.26, No.6, pp.419–429 (1983).
3) Ungar, D. and Jackson, F.: Tenuring Policies for Generation-based Storage Reclamation, *OOPSLA'88*, pp.1–17 (1988).
4) Ungar, D. and Jackson, F.: An Adaptive Tenuring Policy for Generation Scavengers, *ACM Trans. Programming Languages and Systems*, Vol.14, No.1, pp.1–27 (1992).
5) Wilson, P. and Moher, T.: Design of the Opportunistic Garbage Collector, *Proc. OOPSLA'89*, pp.23–35 (1989).
6) 田中詠子，田中良夫，中西正和：Adaptive Garbage Collection の提案及び実験，電子情報通信学会論文誌，Vol.J77-D-I, No.9, pp.611–618 (1994).
7) 田中詠子，田中良夫，中西正和：Adaptive Garbage Collection—実装とその評価，電子情報通信学会論文誌，Vol.J79-D-I, No.5, pp.253–260 (1996).
8) Moon, D.: Garbage Collection in a Large Lisp System, *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pp.235–245 (1984).
9) Chikayama, T., Fujise, T. and Sekita, D.: A Portable and Efficient Implementation of KL1, *PLILP'94* (1994).
10) Fujita, H. and Hasegawa, R.: A Model Generation Theorem Prover in KL1 Using a Ramified-Stack Algorithm. *ICOT TR-606* (1990).
11) Sutter, C., Sutcliffe, G. and Yemenis, T.: The TPTP Problem Library, Technical Re-

port FKI-184-93, Institut für Informatik, Technische Universität München; Technical Report 93/11, Dept. of Computer Science, James Cook University, Townsvill, Australia (1993).

付　　録

今回使用したテストプログラムの紹介

- life.kl1：
  ライフゲーム．幅優先で行うため，ある程度の寿命を持つデータが非常に多い．
- hanoi.kl1：
  ハノイの塔を解くプログラム．手順を生成するフェーズと，それを数えるフェーズの 2 つに分れるが，2 つの実行フェーズ間でゴミ比率に大きな変化はない．長寿命データが比較的多いという特徴がある．
- mastermind.kl1：
  数当てクイズを解くプログラム．単純にすべての解候補を生成し，その後，解のチェックを行う．長寿命データが少なく，データがキャッシュ上に載りやすい．
- kkqueen.kl1：
  N クイーン問題を解くプログラム．最初にあらゆる配置を生成し，その中から正しい解をチェックし，それを数え上げる．解の生成時に大量のデータが作られるため，実行中に 1 度だけゴミ比率に大きな変化が発生する．長寿命データが少なく，データがキャッシュ上に載りやすい．
- MGTPR：
  ボトムアップ型モデル生成型定理証明器による定理証明．与えられた公理系から生成されうるすべての可能世界を深さ優先で探索する．探索が深くなるにつれてゴミが出にくくなるため，ゴミ比率は急増と漸減を繰り返す鋸波状の変化をする．
- MGTPC：
  KL1 プログラムのコンパイル．KLIC 付属の KL1 コンパイラは KL1 で書かれている．コンパイラでは様々な処理がなされるため，ゴミ比率は大きく変動し続ける．

(平成 12 年 3 月 7 日受付)
(平成 12 年 5 月 24 日採録)

吉川　隆英（学生会員）
1974 年生．1999 年東京大学工学系研究科情報工学専攻修士課程修了．現在同博士課程に在学中．日本ソフトウェア科学会，ACM 等会員．

近山　　隆（正会員）
1977 年東京大学卒業．1982 年同大学院博士課程修了．工学博士．同年より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．現在同大学新領域創成科学研究科教授．