

世代別 GC の殿堂入りポリシー：mark/cons 比の限界とスタックフレームからの到達性を利用

林 芳 樹[†] 寺 田 実^{††}

世代別 GC の性能を改善するために、スタックフレームからの到達性を利用した殿堂入りポリシーを提案し、その検証を行った。まず、正確な寿命を持ったアロケーショントレースを使ったシミュレーションにより、殿堂入りポリシーの改善による 2 世代新世代領域固定の世代別 GC の性能の改善の上限を調べた。その結果、殿堂入りポリシーが世代別 GC の性能に影響を与える場合では、mark/cons 比を約半分にまで減少させられる可能性があることが分かった。次に、提案ポリシーの中の 1 つの手法を Java 仮想機械に実装して、シミュレーションでは一番性能の良かった殿堂入りポリシーの実装と比較することで性能を調べた。その結果、提案手法は実装のオーバーヘッドにより実行速度は約 10% 低下したものの、mark/cons 比、メジャーコレクションの回数はほぼ同じか改善されていた。これらの結果から、提案手法は新たな殿堂入りポリシーとして十分に使用可能なものであると結論した。

Tenuring Policy of Generational GC: Ideal Mark/Cons Ratio and a New Policy Using Reachability from Stack Frames

YOSHIKI HAYASHI[†] and MINORU TERADA^{††}

We propose and verify performance of a new tenuring policy based on the reachability from stack frames. First, we measure the collector performance of ideal tenuring policy on 2 generation fixed size nursery generational collector by running simulator against allocation traces with exact life time of objects. We show that mark/cons ratio can be decreased to about half of the best known tenuring policy where tenuring policy matters. Then we implement one of the proposed tenuring policies to Java VM and compare performance with the best tenuring policy in simulation. Proposed policy runs about 10 % slower in total execution time because of runtime overhead but it performs as good as existing tenuring policies in terms of mark/cons ratio and number of major collection. In some cases, proposed policy achieves even better mark/cons ratio. Therefore, we conclude that proposed tenuring policy is as good as existing policies.

1. はじめに

近年、Java, Lisp, ML などガーベジコレクション (GC) 付きのプログラミング言語で書かれたプログラムは急速に増えている。メモリの速度とプロセッサの処理速度の乖離により、効率の良いメモリ管理はますます重要になってきており、GC の性能の改善はプログラム全体の実行速度に大きく影響する。

数ある GC アルゴリズムの中でも非常に重要な手法

に世代別 GC がある。世代別 GC はオブジェクトを複数の世代に分けることにより性能の改善を図る。世代別 GC は頻繁に回収される新世代と、あまり回収されない旧世代からなる。オブジェクトはまず新世代に割り当てられ、ある基準を満たすと旧世代に移動される (tenuring, 殿堂入り)。経験的にほとんどのオブジェクトは短命であることが知られており、この方式はごみの割合が高い領域を集中的に回収することで、非世代別 GC よりもより効率の良いごみの回収を行うことができるようになっている。一般的に、新世代には生きているオブジェクトの量に回収コストが比例するコピー方式 GC⁶⁾ が使用されることが多い。

世代別 GC が効率良く動作するためには、多くのパラメータを正しく設定する必要がある。その中でも重要なパラメータに殿堂入りポリシーがある。殿堂入りが早過ぎると旧世代にごみが堆積する。また殿堂入り

[†] 東京大学大学院情報理工学系研究科知能機械情報学専攻
Department of Mechano-Informatics, Graduate School of Information Science and Technology, The University of Tokyo

^{††} 電気通信大学電気通信学部情報通信工学科
Department of Information and Communication Engineering, Faculty of Electro-Communications, The University of Electro-Communications

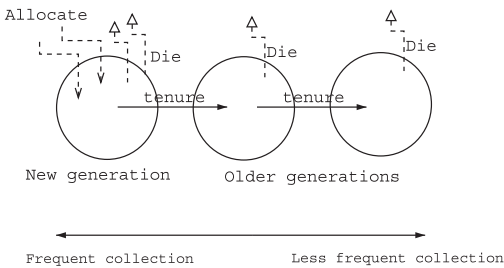


図 1 世代別 GC のヒープ構成

Fig. 1 Heap configuration of generational GC.

が遅過ぎるとオブジェクトの不必要なコピーを繰り返すことになり、どちらも性能の低下を招く。理想的には、長命なオブジェクトを即座に殿堂入りさせ、短命なオブジェクトを殿堂入りさせないのが良い。

本論文では、スタックフレームからの到達性を利用する殿堂入りポリシーを使うことによって、世代別 GC の殿堂入りポリシーの改善を目指す。まず、シミュレーションを利用して、殿堂入りポリシーの改善による性能の向上の限界を調べる。GC 時のオーバヘッドが同じであれば、世代別 GC の性能は mark/cons 比のみで評価できる。したがって、シミュレーションにより定量的な性能の評価を行うことができる。次に、提案手法を Java 処理系に実装し、実際にベンチマークプログラムを実行することにより性能の比較を行う。

2. 世代別コピー方式 GC

世代別 GC は図 1 のように 2 世代や 3 世代にヒープを分割する。オブジェクトは頻繁に回収される新世代に割り当てられ、ある基準を満たすと、あまり回収されない旧世代へ殿堂入りする。世代別 GC には以下の利点がある。

- 新世代領域の効率良い回収
ほとんどのオブジェクトは短命である、という弱世代仮説はほとんどのプログラムについて成立するため、新世代領域はごみの割合が高く、そこを頻繁に回収することで効率を改善する。
- 長命オブジェクトのコピーの回避
長命オブジェクトを旧世代に移動させ、旧世代の回収を少なくすることで長命オブジェクトをコピーするコストを減らす。
- 停止時間の減少
世代別 GC は新世代を頻繁に回収し、ときどきヒープ全体を回収する。新世代は普通あまり大きくないので、GC のほとんどを占める新世代の回収時の停止時間を短くすることができる。

2.1 殿堂入りポリシー

適応的な手法を使うもの以外の既存の殿堂入りポリシーは、GC の経験回数によって殿堂入りを決定する。

- GC を 1 回経験したオブジェクトを殿堂入り
 - － 長所
 - * 長命オブジェクトが何度もコピーされることはない。
 - * 新世代の semispace は 1 面だけでよい。
 - * ヒープ全体の使用が早まるため、メモリを有効に利用できる可能性がある。
 - － 短所
 - * 旧世代領域が早く埋まるため、メジャーコレクションの回数が増える。
- GC を 2 回以上経験したオブジェクトを殿堂入り
 - － 長所
 - * 短命オブジェクトが殿堂入りしなくなるため、メジャーコレクションの回数が少なくなる。
 - － 短所
 - * 長命オブジェクトは殿堂入りするまでに、必ず指定回数まで新世代領域でコピーされる。

3. 提案するポリシー：スタックからの到達性の利用

既存の殿堂入りポリシーでは、殿堂入りが早過ぎたり、遅過ぎたりすると考えられる。本章では、スタックフレームからの到達性を利用した新しい殿堂入りポリシーを提案する。

スタックからの参照はルートセットの一部であるため、仮にオブジェクトへの参照が書き換えられることがないとなると、スタックの動作原理から、より深いフレームから指されているオブジェクトはより浅いフレームから指されているオブジェクトより必ず長命になる。したがって、参照が書き換えられることのある環境でもどのスタックフレームから到達可能か、という情報を用いてオブジェクトの寿命をある程度正確に予測できると予想できる。また、静的領域はつねに生きているオブジェクトの存在する領域なので、そこから指されているオブジェクトは死なないか、長命であることが予想される。これらをもとに、オブジェクトの寿命について以下の仮説をたてる。

- 静的領域から到達可能なオブジェクトはほとんど死なない。
- 比較的浅いスタックフレームからのみ到達可能なオブジェクトはほぼ短命である。

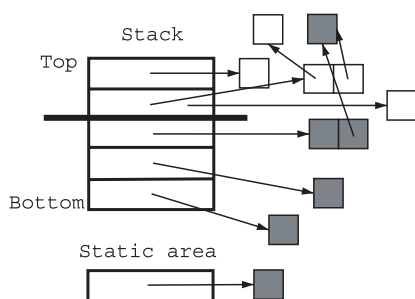


図 2 スタックフレームからの到達性を用いた殿堂入り

Fig. 2 Tenuring policy based on reachability from stack frames.

- 比較的深いスタックフレームから到達可能なオブジェクトは短命ではない。

これを 2 世代別 GC に適用すると、図 2 のように、あるスタックフレームよりも深い領域から指されているオブジェクト（灰色）は殿堂入りさせ、それ以外のオブジェクト（白色）は新世代にとどめる、というようになる。

3.1 フレームの分割と世代の設定

この提案手法では殿堂入りを分けるスタックフレームの決定方法が非常に重要になる。たとえば、以下のような方法が考えられる。

- 2 世代別 GC
 - スタックの一定の割合以下のフレームから到達可能なオブジェクトは殿堂入りさせる。
 - 前回の GC のときに存在していたフレームから到達可能なオブジェクトは殿堂入りさせる。
- 3 世代別 GC
 - Pre-tenuring^{1),4),7),9)} 風 3 世代 GC
 - * 第 1 世代

前回の GC のときに存在していたフレームから到達可能なオブジェクトは第 3 世代へ移動、残りは第 2 世代へ移動させる。
 - * 第 2 世代

GC を生き残ったオブジェクトは第 3 世代へ移動させる。

本論文では、2 世代の、前回の GC のときに存在していたフレームから到達可能なオブジェクトを殿堂入りさせる方法を評価する。

4. シミュレーションによる mark/cons 比の測定手法

シミュレーションによる方法には、Hansen ら⁸⁾ などに使用されている synthetic ベンチマークにより実際のプログラムに似せたアロケーションデータを生成す

る方法と、実際のプログラムの実行から取得したアロケーショントレースを用いて行う方法がある。本論文では、後者のアロケーショントレースを用いてシミュレーションを行う。このための方法には Merlin アルゴリズム¹⁰⁾ を用いる。

アロケーショントレースには、オブジェクトが生成された時刻とサイズ、死亡した時刻、死亡したときに指していたオブジェクトが記録されており、ファイル中のオブジェクトはアロケーションされた順に並んでいる。ここでの時刻は、オブジェクトがアロケーションされる前までの総アロケーション量で表される。

シミュレータは、アロケーショントレースを前から順に処理していき、ヒープが足りなくなれば GC を行う。オブジェクトがごみであるかどうかは、GC を起動したオブジェクトの時刻と、回収対象になっているオブジェクトの死亡時刻を比較することで調べることができる。

ただし、世代別 GC のマイナーコレクションにおいては、旧世代からの参照はルートセットとして扱うため、トレース上は死んでいるが、生きているものとして扱うオブジェクトも存在する。したがって、世代別 GC のシミュレーションは世代別でない GC よりも複雑になる。オブジェクトがごみになった時点で指していたオブジェクトが記録されているのはこの処理を実現するためである。

2 世代別 GC のシミュレーションは次のように行う。

- マイナーコレクション
 - 旧世代領域のオブジェクトは生存させる。
 - 新世代領域のトレース上で生きているオブジェクトは生存させる。
 - 旧世代領域のごみオブジェクトをルートセットとし、そこからごみオブジェクト間の参照をたどっていつて到達可能な新世代領域のオブジェクトは生存させる。
 - それ以外のオブジェクトはごみとする。
- メジャーコレクション
 - 新世代、旧世代ともにトレース上で生きているオブジェクトは生存させる。
 - それ以外はごみとする。

5. 結 果

本章では、シミュレーションの結果から、殿堂入りポリシーの改良による世代別 GC の性能改善の限界を示す。次に、実際に提案手法の 1 つを実装し、実行速度と mark/cons 比、メジャーコレクションの回数に

表 1 ベンチマーク

Table 1 Benchmark programs.

ベンチマーク	内容
_201_compress	Lempel-Ziv (LZW) を使用した圧縮
_202_jess	エキスパートシステム
_209_db	データベースのシミュレーション
_227_mtrt	マルチスレッドレイトレーサ
_228_jack	パーザ生成器

について性能を評価する。

5.1 評価方法

● Jikes RVM

シミュレーションのためのトレースの採取と、プログラムの実行による性能の比較には、Java 処理系である Jikes RVM 2.1.0^{2),3)} を使用した。

Jikes RVM は Java で書かれた JIT コンパイラのみからなる処理系であり、アプリケーションと VM がシームレスに動作し、Jikes RVM 自身も Jikes RVM でバイトコードからマシンコードに変換される。つまり、GC の性能の改善はアプリケーションのみならず、VM の性能にも影響する。コンパイラには、バイトコードをほぼそのままマシンコードに変換する Base コンパイラと、中間表現にしてからいろいろな最適化をかける Opt コンパイラがある。評価には、Jikes RVM もアプリケーションも Base コンパイラを使ってコンパイルする VM を使用した。

● ベンチマーク

SPECjvm98 の 7 つのプログラムの中から表 1 の 5 つのプログラムをベンチマークとして使用した。SPECjvm98 のベンチマークはアロケーションの量があり多くなく、典型的なアプリケーションの振舞いを網羅しているわけでもないため、GC のためのベンチマークとしては理想的なものではないとされている。しかし、代わるベンチマークも存在しないため、Java 言語の処理系における GC の研究では頻繁に用いられている。なお、_222_mpegaudio はほとんどアロケーションを行わないので、_213_javac は動作させることができなかったため、今回の実験対象から除外した。

● ハードウェア実行速度の測定は、Pentium III 800 MHz、メモリ 640 MB の環境で行った。

5.2 シミュレーションによる mark/cons 比の測定結果

以下の殿堂入りポリシーのシミュレーションを行った。

- (1) n 回 ($1 \leq n \leq 3$) GC を経験したオブジェクトを殿堂入りさせる 2 世代の世代別 GC

表 2 シミュレーション用のアロケーショントレースを収集したベンチマーク

Table 2 Benchmark programs used to gather allocation traces for GC simulation.

ベンチマーク	最小ヒープサイズ (MB)	総アロケーション (MB)
_201_compress	19	115,164,648
_202_jess	10	283,228,060
_209_db	22	82,755,464
_228_jack	15	266,502,584

- (2) 余命が新世代領域の 0.5 倍、1 倍、2 倍内のオブジェクト以外を殿堂入りさせる 2 世代の世代別 GC

後者は完全な寿命予測に基づいた殿堂入りポリシーであり、そのときの性能は理想的な状態での性能と見なすことができる。図中では、前者を 2g 1、2g 2、2g 3 というように表示し、後者を 2gp 0.5、2gp 1、2gp 2 というように表示する。

シミュレーションには、表 2 のベンチマークから取得したアロケーショントレースを使用した。ここで _227_mtrt を含めていないのは、mtrt はマルチスレッドアプリケーションであるため、スレッド切替えのタイミングによってトレースの内容が異なり、シミュレーションの結果と実際の実行の結果が必ずしも一致しないからである。最小ヒープサイズは GC を 1 回で殿堂入りさせるポリシーの実行に最小限必要なサイズである。各ベンチマークは “rvm SpecApplication -s100 -m1 -M1 _201_compress” のようにして実行した。

新世代領域はヒープサイズの 20%、メジャーコレクションは旧世代領域の空き容量が新世代領域のサイズよりも小さくなったときに行われる。たとえば、ヒープサイズが 10 MB のときは新世代領域は 2 MB、旧世代領域は 4 MB + 4 MB になり、旧世代領域の semispace の残りが 2 MB 以下になったときの GC はメジャーコレクションになる。すべての殿堂入りポリシーにおいて、アロケーションに使用できるメモリの量が同じになるようにしてシミュレーションを行った。GC 1 回で殿堂入りさせるポリシー以外は、新世代領域にもう 1 面 semispace が必要なため全体ではより多くのメモリを使用している。たとえば、グラフ中で 10 MB と表示されているところでは、GC 1 回のポリシーは新世代領域 2 MB、旧世代領域 4 MB + 4 MB で 10 MB を、それ以外のものは新世代領域 2 MB + 2 MB、旧世代領域 4 MB + 4 MB で 12 MB を使用している。

5.2.1 mark/cons 比

図 3 は各ベンチマークにおける mark/cons 比であ

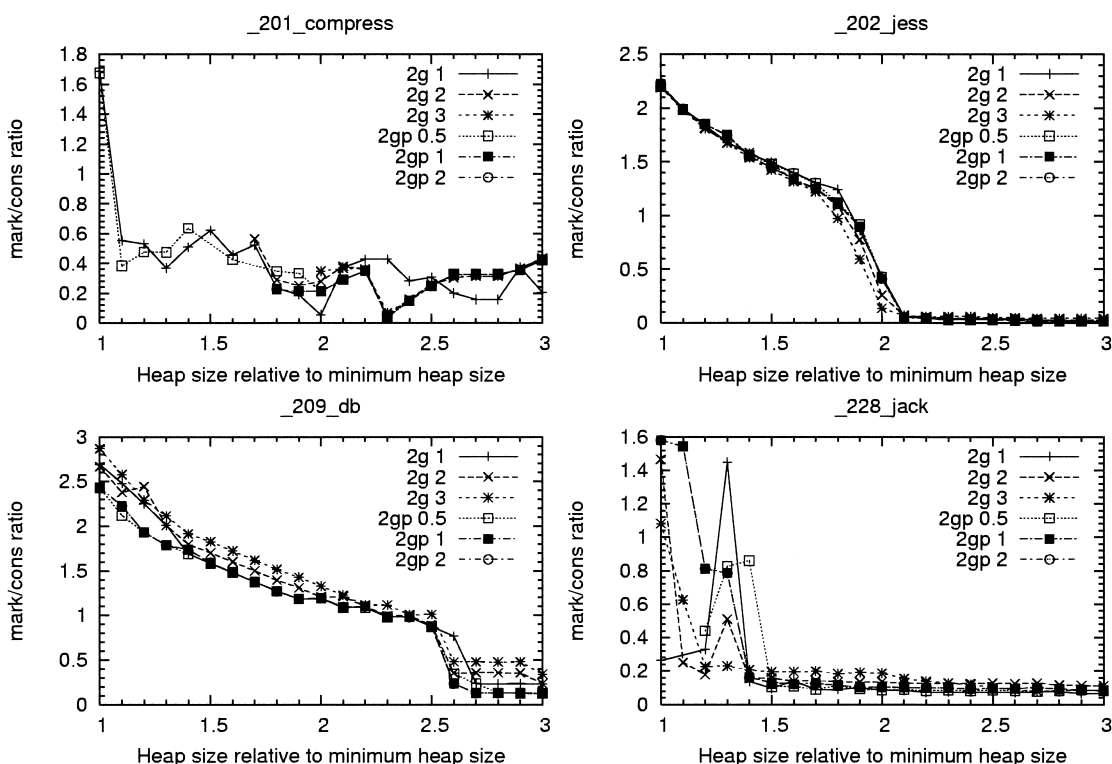


図 3 シミュレーションによる各殿堂入りポリシーの mark/cons 比

Fig. 3 Simulated mark/cons ratio of tenuring policies.

る。メジャーコレクションの回数を図 4 に示す。

図 3, 4 より, 2 つのグラフはほぼ同じ形をしていることが分かる。メジャーコレクションとマイナーコレクションの回数の関係により, グラフは図 5 のように以下の 4 つの区域に分割できる。

- 区域 1: GC のほとんどをメジャーコレクションが占める区域。
- 区域 2: メジャーコレクションとマイナーコレクションの回数がほぼ同じ区域。
- 区域 3: GC のほとんどをマイナーコレクションが占める区域。
- 区域 4: GC のすべてをマイナーコレクションが占める区域。

各区域には以下の特徴がある。

- 区域 1 は, ほぼ semispace GC と同じ振舞いをする。ここでは, ほとんど毎回メジャーコレクションが起こるため, なるべく GC が起こるのを遅くするために, 殿堂入りはなるべく早い方がよい。
- 区域 2 は, 区域 1 と区域 3 が組み合わさった区域である。プログラムの実行の最初の方はほぼマイナーコレクションばかりであるが, プログラムのワーキングセットが徐々に増えていき, 旧世代

領域の一定の割合を占めるようになって, それ以後はつねにメジャー GC が起こる。殿堂入りポリシーの違いによる mark/cons 比への影響は様々である。

- 区域 3 は, 殿堂入りポリシーによって性能が変化する区域である。
- 区域 4 は, メジャーコレクションが起こらない区域であるから, 新世代領域をなるべく有効に使うために, 殿堂入りはなるべく早い方がよい。

区域 1, 2 は世代別 GC が有効に動作しない区域である。区域 4 は, プログラムのアロケーションが多ければ区域 3 に含まれるため, ある程度長い時間実行されるプログラムでは起こらない区域だと考えられる。したがって, 世代別 GC が有効に動作する区域 3 で性能が良くなることが重要である。図 6 に, 区域 3 だけを拡大したものを示す。

図 6 から以下のことが分かる。

- _201_compress と _228_jack では, 正確な寿命予測を用いた殿堂入りポリシーと GC 経験回数に基づく殿堂入りポリシーの中で一番効率が良いものがほぼ同じ性能であり, _202_jess と _209_db では, 正確な寿命予測を用いた殿堂入りポリシーの

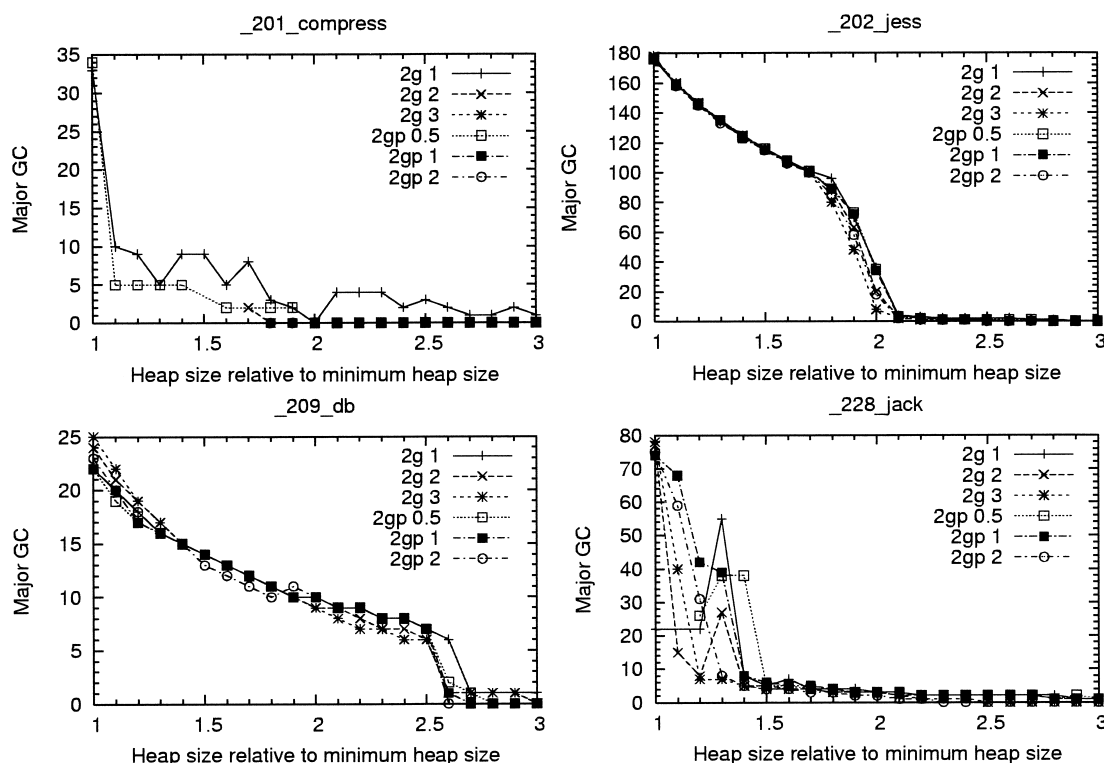


図 4 シミュレーションによる各殿堂入りポリシーのメジャーコレクションの回数

Fig. 4 Simulated number of major collections of tenuring policies.

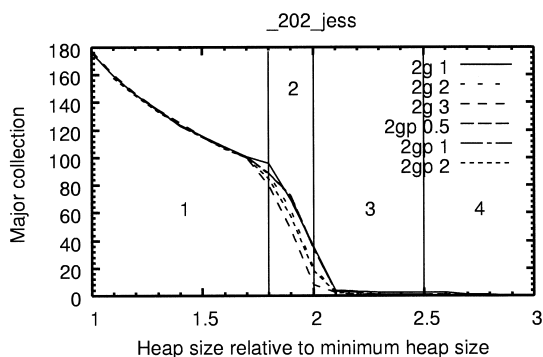


図 5 メジャーコレクションの割合による区域の分類

Fig. 5 Classify heap sizes by percentage of major collection.

方が性能が良い。

- **_201.compress** 以外では、GC 回数に基づいた殿堂入りポリシーは殿堂入りが早い方が効率が良い。
- 正確な寿命予測に基づいた殿堂入りポリシー間にはほぼ効率に差がない。

5.3 提案手法の実行結果

GC を 1 回経験したオブジェクトを殿堂入りさせる Jikes RVM 付属の 2 世代別 GC を変更した copyGen コレクタと、前回の GC から存在したスタック

表 3 ガーベジコレクタ

Table 3 Garbage collector.

コレクタ	copyGen (オリジナル)	stackGen (提案手法)
殿堂入りポリシー	GC 1 回	フレームからの到達性
メジャーコレクション	旧世代残り 512 Kb	旧世代残り 512 Kb
新世代領域	1 面	2 面
旧世代領域	2 面	2 面
System.gc	なし	なし
Large Object Area	なし	なし

クフレームから到達可能なオブジェクトを殿堂入りさせるポリシーを実装した stackGen コレクタの性能の比較を行った。各コレクタの特徴は表 3 のようになっている。

stackGen コレクタの殿堂入りを分けるフレームの選択方法は、前回の GC から存在するフレームから到達可能なオブジェクトは殿堂入りという方法を用いた。

スタックフレームの変化を追跡する方法はいろいろあるが、どのような場合でも効率良く追跡するための方法はよく分かっていない。ただし、提案した殿堂入りポリシーでは、各スタックフレームが前回の GC 時に存在したか否かを判別できればよいため、その情報

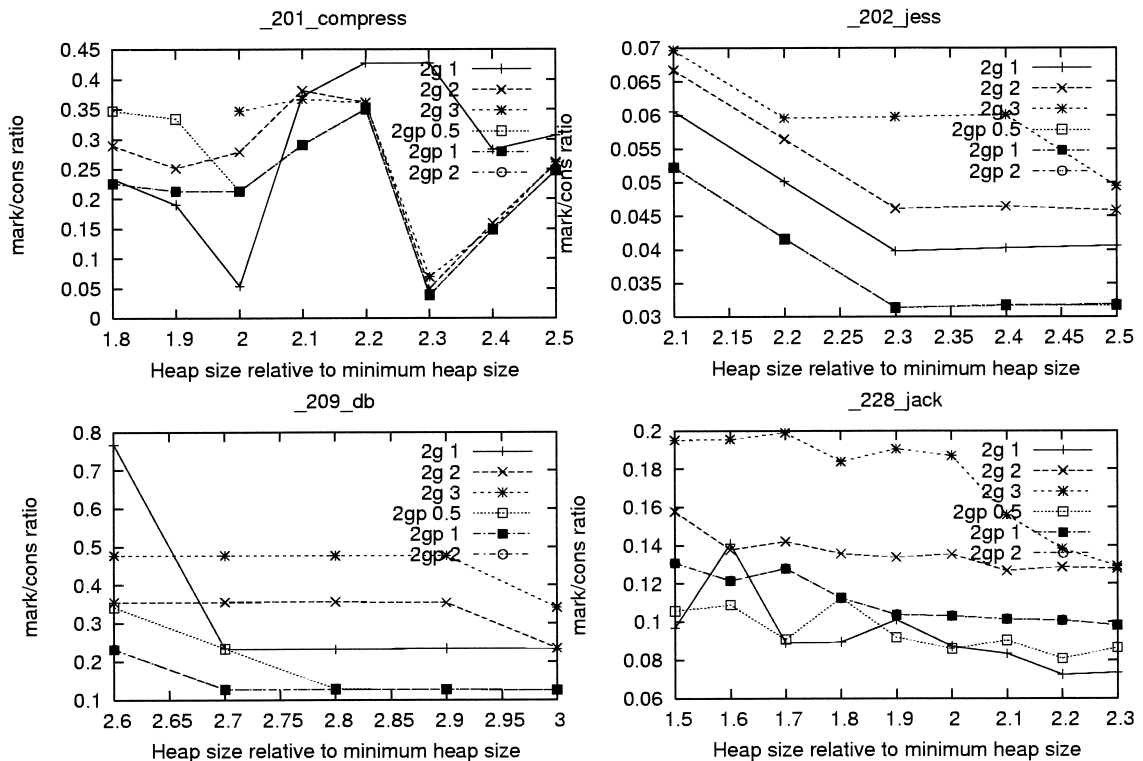


図 6 区域 3 におけるシミュレーションによる各殿堂入りポリシーの mark/cons 比

Fig. 6 Simulated mark/cons ratio of tenuring policies within region 3.

を格納するための領域を各スタックフレームに追加し、そこに情報を格納することで代用することができる。

今回の実装においては、スタックフレームのフレームポインタなどが存在するヘッダ領域を 4 バイト拡張し、そこが 0 である場合は GC 後に生成されたフレーム、1 である場合は GC 前に存在するフレームとして区別している。これは、スタックフレームの生成時と、GC 時にルートセットを数えあげるために各スタックフレームをたどって参照を処理するときに新たな処理を追加することでやっている。つまり、

- スタックフレームの生成時に判別用のフィールドに 0 を設定する、
 - GC 時に各スタックフレームを処理するときに、判別用のフィールドに 1 を設定する、
- ということを行っている。この方式はスタックの深さが減るときの処理を考慮しなくてよいため、例外による非局所脱出への対応が非常に簡単になっている。

性能の測定には表 4 のベンチマークを使用した。最小ヒープサイズは copyGen コレクタの実行に必要なヒープのサイズである。シミュレーションのときと同様、アロケーションに使用できるメモリの量が同じになるようにした。stackGen コレクタは新世代領域で

表 4 提案手法の評価のために使用したベンチマーク

Table 4 Benchmark programs used to evaluate proposed tenuring policy.

ベンチマーク	最小ヒープサイズ (MB)
_201_compress	25
_202_jess	8
_209_db	20
_227_mtrt	17
_228_jack	8

semispace を 1 面余分に使用するので、使用ヒープサイズは copyGen コレクタよりもその分多くなっている。

各ベンチマークプログラムにおける実行速度は図 7 のようになった。1 から 2 までは 0.1 間隔、2 から 3 までは 0.5 間隔で測定した。点が抜けているのは指定のヒープサイズではメモリが足りなくなり、最後までベンチマークを実行できていないところである。各ベンチマークは 5 回実行をし、一番速いものと遅いものを除いた 3 回の平均を実行速度として用いた。新世代領域はメガバイト単位での指定になっており、たとえば、ヒープサイズが 25 MB のときの新世代領域の大きさは 6 MB である。

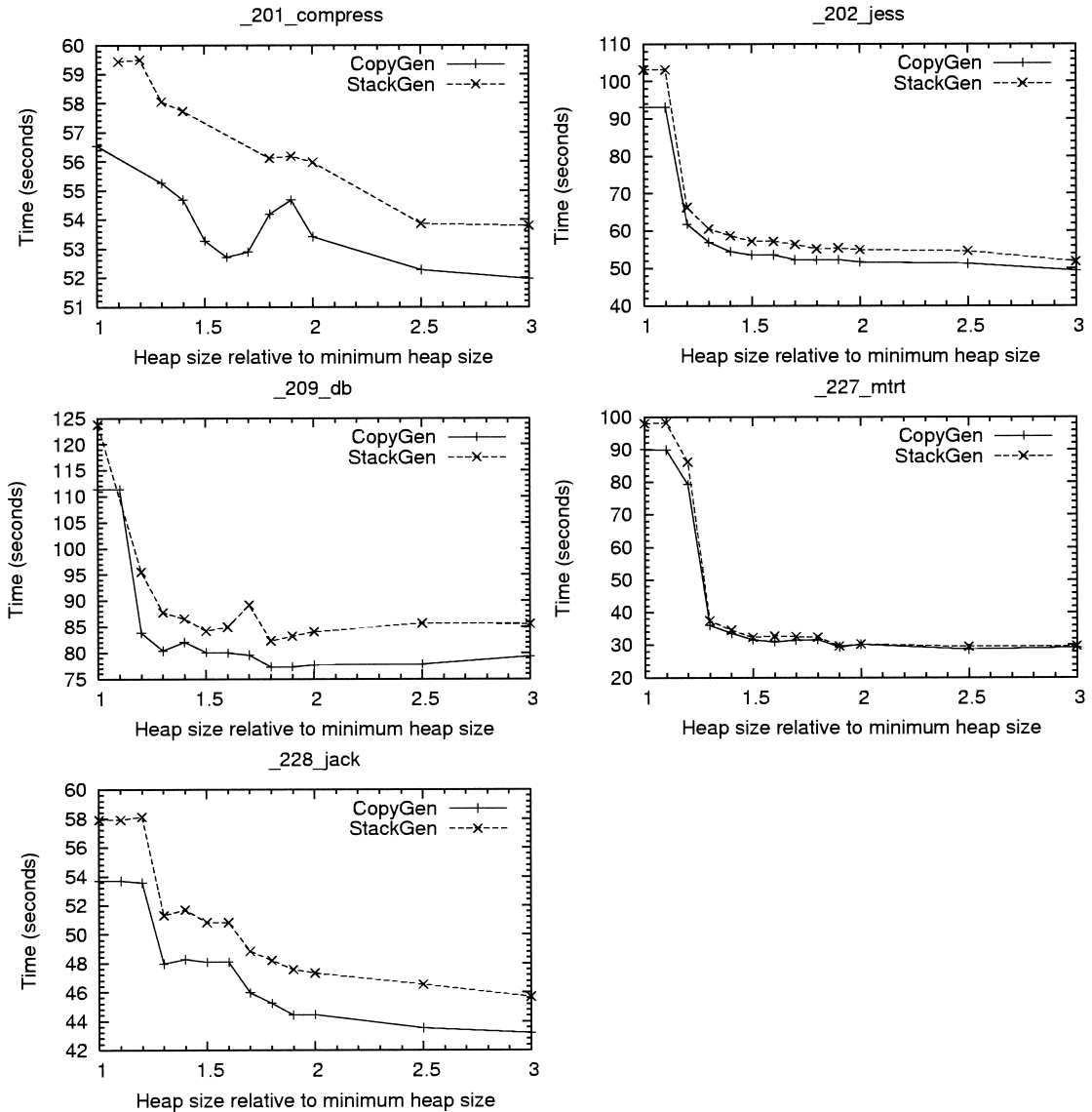


図 7 プログラム全体の実行速度
Fig. 7 Total application performance.

時間は SPECjvm98 が表示する値を使用した。各ベンチマークは、“rvm SpecApplication -s100 -m1 -M1 -201_compress” のようにして、コマンドラインから実行した。

図 7 がプログラム全体の実行速度の結果である。これより、すべてのベンチマークで提案手法の方が実行速度は遅くなってしまっていることが分かる。

各ベンチマークプログラムの mark/cons 比は図 8 のようになった。_201_compress 以外では mark/cons 比はほぼ同じになっている。

図 9 に各々の新世代領域に対するメジャーコレク

ションの回数を示す。

_201_compress を除いては、ヒープの大きさが小さいときは copyGen の方がメジャーコレクションの回数は少ないが、ある程度メジャーコレクションの数が少なくなっている領域に関しては、copyGen と stackGen のメジャーコレクションの回数はほぼ同じであることが分かる。

6. 関連研究

Hirzel ら¹¹⁾ は、オブジェクトを参照している場所と寿命の関係を調べた。その結果、スタックからしか

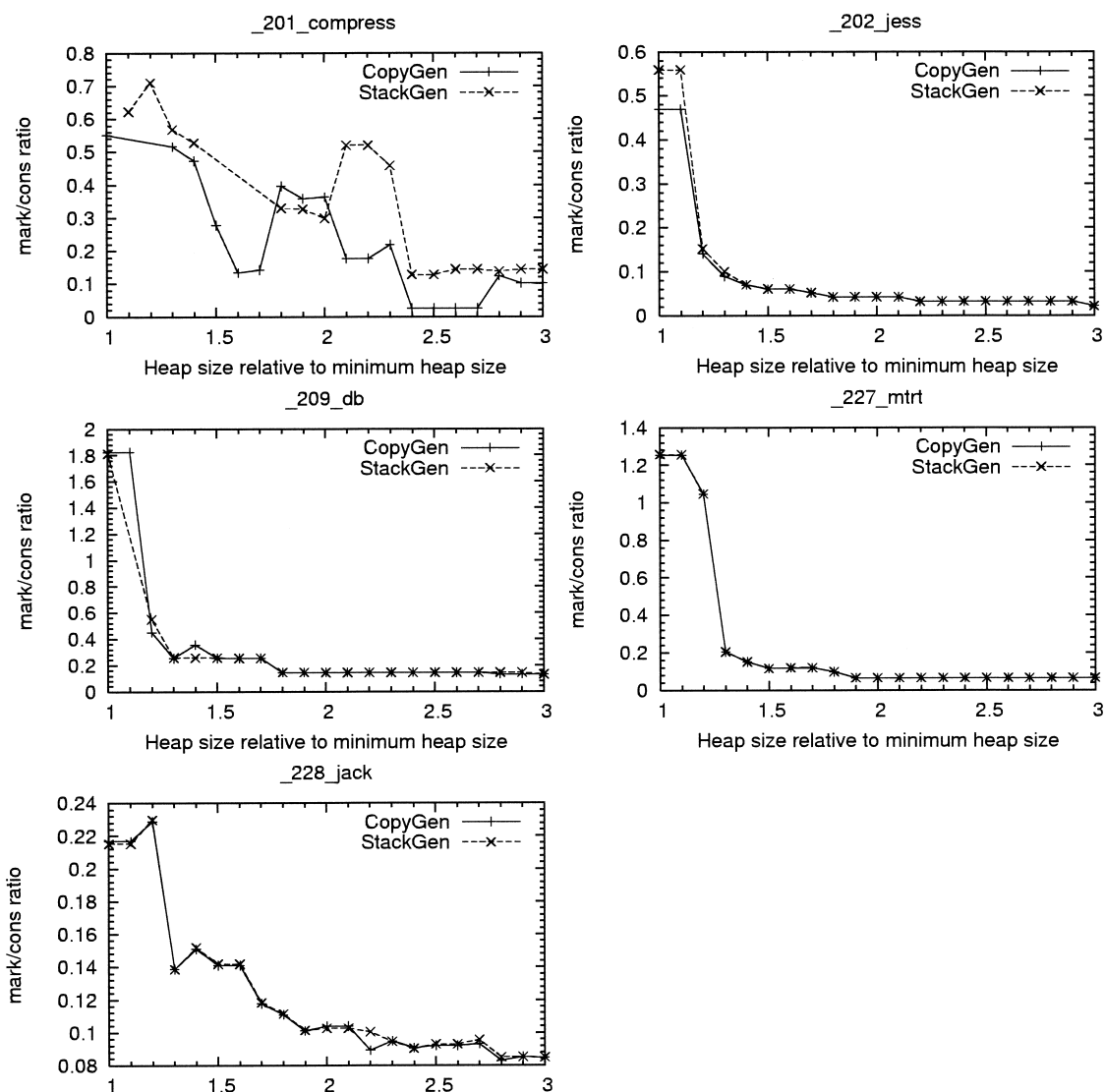


図 8 mark/cons 比
Fig. 8 mark/cons ratio.

指されていないオブジェクトは大部分が短命であり、グローバルから到達可能なオブジェクトはほぼ死なないオブジェクトであることを示した。

Cannarozzi ら⁵⁾は、ポップされたスタックフレームからしか参照できないオブジェクトはごみである性質を利用した、スタックフレームからの到達性のみを用いた GC アルゴリズムを提案した。

短命オブジェクトの殿堂入りを避ける代わりに、そもそもアロケーションされたばかりのオブジェクトは GC の対象外にする方法に older-first GC^{12)~14)}がある。

長命オブジェクトのコピーを減らす方法には、長命

になりそうなオブジェクトをアロケーション時に旧世代に割り当てる pre-tenuring がある。長命オブジェクトの予想には、以前のプログラムの実行を基にしたプロファイルによるもの^{4),7),9)}とプログラムの実行時に標本を採取して決定するもの¹⁾がある。

7. 結 論

本論文では、まず、理想的な殿堂入りポリシーを使用するとプログラムによっては約半分にまで mark/cons 比を改善可能なことを示した。

次に、スタックフレームからの到達性を利用した殿堂入りポリシーを提案し、その中の 1 つの手法を実際

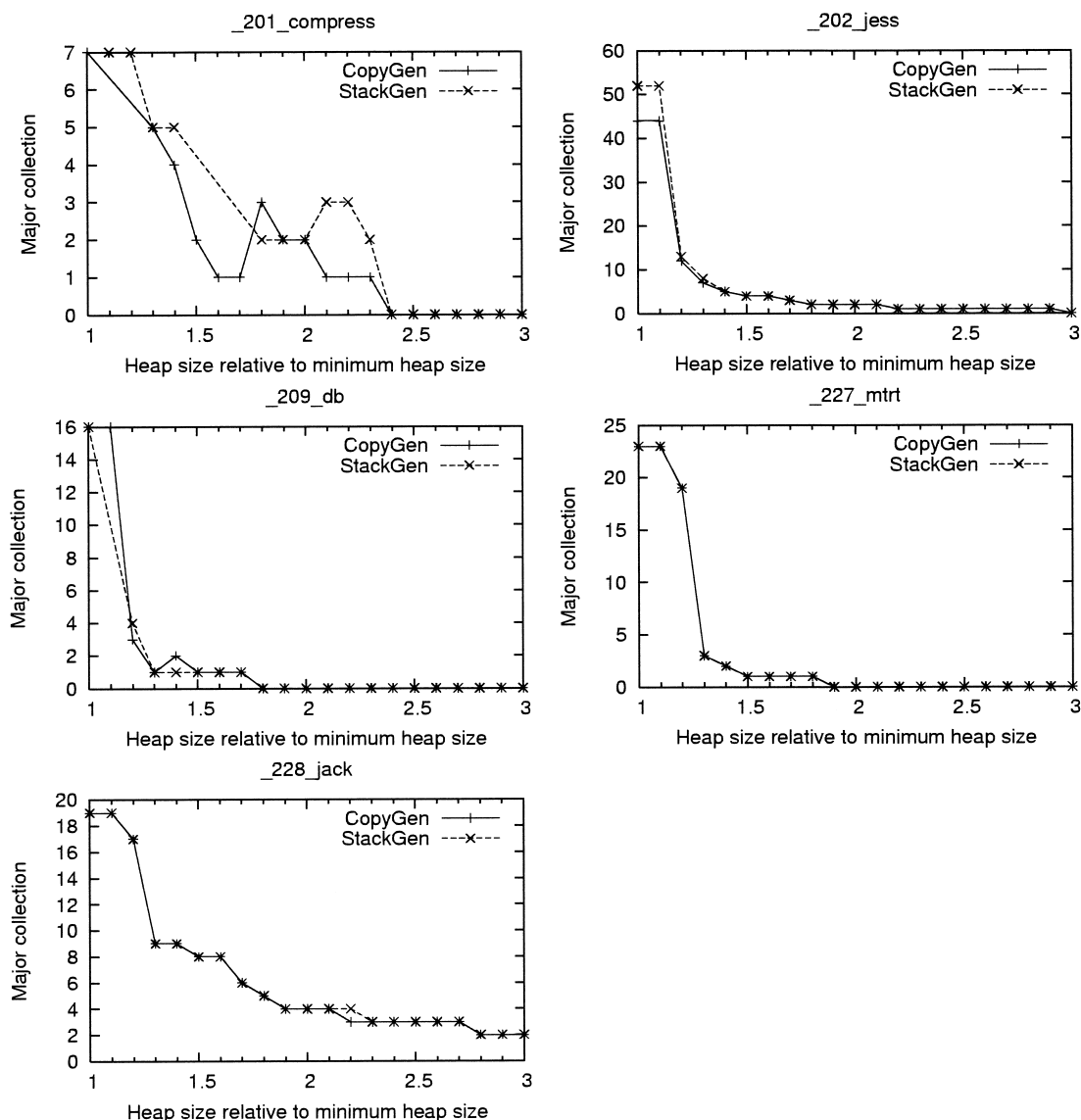


図9 メジャーコレクションの回数
Fig.9 Number of major collection.

に Java 処理系に実装し、シミュレーションでは既存の殿堂入りポリシーの中で一番良い性能を出した GC を 1 回経験したオブジェクトを殿堂入りさせるポリシーと性能を比較した。その結果、実装のオーバーヘッドによりプログラム全体の速度は 10% 程低下したが、mark/cons 比、メジャーコレクションの回数はともにほぼ変わらず、提案手法の方が良くなる場合もあることが分かった。

実装のオーバーヘッドを減らし、より良いスタックフレームの選択をすることができれば既存の殿堂入りポリシーよりも性能が良くなる可能性もあり、スタック

フレームからの到達性を利用した殿堂入りポリシーは十分に可能性のある手法であるといえる。

参考文献

- 1) Agesen, O. and Garthwaite, A.: Efficient object sampling via weak references, *Proc. 2nd International Symposium on Memory Management*, pp.121–126 (2000).
- 2) Alpern, B., Attanasio, C.R., Cocchi, A., Lieber, D., Smith, S., Ngo, T., Barton, J.J., Hummel, S.F., Sheperd, J.C. and Mergen, M.: Implementing jalapeño in Java, *Proc.*

- 1999 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pp.314–324 (1999).
- 3) Alpern, B., Attanasio, D., Barton, J.J., Burke, M.G., Cheng, P., Choi, J.-D., Cocchi, A., Fink, S.J., Grove, D., Hind, M., Hummel, S.F., Lieber, D., Litvinov, V., Mergen, M., Ngo, T., Russell, J.R., Sarkar, V., Serrano, M.J., Shepherd, J., Smith, S., Sreedhar, V.C., Srinivasan, H. and Whaley, J.: The Jalapeño Virtual Machine, *IBM System Journal*, Vol.39, No.1, pp.211–238 (2000).
 - 4) Blackburn, S.M., Singhai, S., Hertz, M., McKinley, K.S. and Moss, J.E.B.: Pretenuring for Java, *Proc. OOPSLA '01 Conference on Object Oriented Programming Systems Languages and Applications*, pp.342–352 (2001).
 - 5) Cannarozzi, D.J., Plezbert, M.P. and Cytron, R.K.: Contaminated garbage collection, *Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp.264–273 (2000).
 - 6) Cheney, C.J.: A Non-Recursive List Compacting Algorithm, *Comm. ACM*, Vol.13, No.11, pp.677–678 (1970).
 - 7) Cheng, P., Harper, R. and Lee, P.: Generational stack collection and profile-driven pretenuring, *Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pp.162–173 (1998).
 - 8) Hansen, L.T. and Clinger, W.D.: An experimental study of renewal-older-first garbage collection, *ACM SIGPLAN Notices*, Vol.37, No.9, pp.247–258 (2002).
 - 9) Harris, T.L.: Dynamic adaptive pre-tenuring, *Proc.2nd International Symposium on Memory Management*, pp.127–136 (2000).
 - 10) Hertz, M., Blackburn, S.M., McKinley, K.S., Moss, J.E.B. and Stefanovic, D.: Error-Free Garbage Collection Traces: How to Cheat and Not Get Caught, *Proc. International Conference on Measurements and Modeling of Computer Systems*, pp.140–151 (2002).
 - 11) Hirzel, M., Henkel, J., Diwan, A. and Hind, M.: Understanding the connectivity of heap objects, *Proc. 3rd International Symposium on Memory Management*, pp.36–49 (2002).
 - 12) Stefanović, D.: Properties of Age-Based Automatic Memory Reclamation Algorithms, Ph.D. Thesis, University of Massachusetts (1999).
 - 13) Stefanović, D., Hertz, M., Blackburn, S.M., McKinley, K.S. and Moss, J.E.B.: Older-first Garbage Collection in Practice: Evaluation in a Java Virtual Machine, *Memory System Performance*, Berlin, Germany (2002). <ftp://ftp.cs.umass.edu/pub/osl/papers/msp-2002-of-main.ps.gz>.
 - 14) Stefanović, D., McKinley, K.S. and Moss, J.E.B.: Age-based garbage collection, *Proc. 1999 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp.370–381 (1999).

(平成 15 年 2 月 18 日受付)

(平成 15 年 7 月 9 日採録)



林 芳樹

1978 年生。2001 年東京大学工学部電子情報工学科卒業，同年同大学院情報理工学系研究科知能機械情報学専攻修士課程入学。2003 年同修士課程修了，同年同大学院情報理工学系研究科電子情報学専攻博士課程入学。現在，同博士課程在学中。プログラミング言語処理系に興味を持つ。



寺田 実 (正会員)

1959 年生。1981 年東京大学工学部計数工学科卒業。1983 年同大学院工学系研究科情報工学修士課程修了。同大学計数工学科助手，電気通信大学電子情報学科助手をへて 1991 年東京大学工学部機械情報工学科講師，1992 年同大学助教授。2002 年電気通信大学電気通信学部情報通信工学科助教授，現在に至る。工学博士。主な研究分野はプログラム言語処理系，プログラミング環境等。ソフトウェア科学会，ACM 会員。