Rubyにおけるライトバリアのないオブジェクトを考慮した世代別インクリメンタルGCの実装

○笹田 耕一[1,a)]　松本 行弘[1,b)]

概要：
ガーベージコレクション（GC）の改善手法として世代別 GC、およびインクリメンタル GC はよく知られているが、実装するためには正確なライトバリアの挿入が必須である。一方、Ruby 処理系は保守的マーク&スイープ GC を用いることで、C 言語などで拡張コードを書くときに余計なコードを含む必要がなかったが、ライトバリアを挿入しておらず、これを必要とする GC アルゴリズムを導入することができなかった。我々はこの問題を解決するために、ライトバリアに対応しているオブジェクトとしていないオブジェクトを区別して、世代別インクリメンタル GC を実装する新しい方法を提案する。本発表では提案する手法について説明し、Ruby への実装について述べる。

Implementation of a generational incremental garbage collector with write barrier unprotected objects on Ruby

Koichi Sasada[1,a)]　Yukihiro Matsumoto[1,b)]

Abstract:
Generational GC and incremental GC are well known algorithms to improve performance of garbage collector, and they require correct write barriers in the interpreter. On the other hands, Ruby interpreter uses conservertive mark and sweep GC, and we can implement C extensions easily without any extra codes. However, Ruby interepreter does not have any write barriers and we can not implement GC algirithms which require correct write barriers. To solve this problem, we propose new technique to implement generational and incremental GC by separating all objects into write barrier protected objects and write barrier unprotected objects. In this presentation, we will describe new technique and an implementation on Ruby interpreter.

1. はじめに

オブジェクト指向プログラミング言語 Ruby[3] は、ウェブアプリケーション開発などの分野で世界中で広く利用されている。Ruby で書かれたプログラムを実行するためのインタプリタはいくつか存在するが、我々が開発している Ruby インタプリタがもっとも多く利用されている [4][*1]。本稿では、Ruby 処理系と単に呼ぶとき、この我々が開発している Ruby インタプリタを指すこととする。

2004 年ごろから、Ruby で記述されたウェブアプリケーション開発フレームワークである Ruby on Rails などが登場し、例えば、初期のバージョンの Twitter や、github など、多くのアプリケーションが Ruby で開発されるようになった。このような背景から、Ruby 処理系に対する性能の要求は高くなってきている。

Ruby 処理系の性能にはいくつか改善を要する要素があるが、その 1 つにガーベージコレクション（GC）があげられる。GC は、他のオブジェクト指向プログラミング言語と同様に、不要となったオブジェクトを自動的に回収し、

[1] Heroku, Inc.
[a)] ko1@heroku.com
[b)] matz@heroku.com

[*1] MRI や CRuby とも呼ばれる。他には、Java で開発された JRuby や、インタプリタの大部分を Ruby 自体で記述した Rubinius がある。

メモリなどの計算機資源を再利用可能にするための機構である。Ruby 処理系は、1993 年の開発当初から、保守的マークアンドスイープを利用してきた。しかし、Ruby が用いる保守的マークアンドスイープは、多くのオブジェクトが存在するとき、処理性能が低く、停止時間が長いという問題がある。

例えば Ruby がよく利用されるウェブアプリケーションの分野では、処理性能が遅いと単位時間あたりに処理することができる HTTP リクエスト数が減り、必要な計算機リソースが増えるという問題がある。また停止時間が長いと 1 リクエストにかかる時間が増え、エンドユーザの待ち時間が増え、問題となる。

処理性能を向上する方法として世代別 GC、停止時間を短くする手法としてインクリメンタル GC が知られている [1]。しかし、これらの手法ではライトバリアという、オブジェクトへの書き込みを検知する手法を実装する必要がある。

Ruby 処理系は C 言語で実装されているが、保守的マークアンドスイープアルゴリズムを前提としているため、その実装にはライトバリアが一切入っていなかった。また、Ruby 処理系を C 言語で拡張する拡張ライブラリでも、ライトバリアが無い。Ruby 処理系への適切なライトバリア挿入は、不可能ではないが、ミスが許されない作業であり、高い開発コストが必要になる。また、拡張ライブラリへのライトバリアの導入は、既存の拡張ライブラリに非互換をもたらし、これまで開発されてきた C 拡張ライブラリが利用できなくなるという問題がある。そのため、世代別 GC やインクリメンタル GC が実装できなかった。

我々は、この問題を解決するため、ライトバリアのないオブジェクトを考慮した世代別 GC、およびインクリメンタル GC を考案し、Ruby 処理系に実装した。考案したアルゴリズムでは、ライトバリアに対応していないオブジェクトがあっても適切に動作するため、既存の拡張ライブラリがそのまま動作する。

考案した世代別 GC は、従来の手法よりも若干制限があり、性能が低くなる可能性があるため、Restricted Genrational GC（RGenGC）と名付けた。同様に、考案したインクリメンタル GC は従来の方式よりも停止時間が長くなることがあるので、Restricted incremental GC（RincGC）と名付けた。

評価の結果、制限がありながらも、これまでの保守的マークアンドスイープ GC に比べ、処理性能が向上し、停止時間が短くなったことが確認出来た。

なお、考案した世代別 GC アルゴリズムを導入した Ruby 処理系を 2013 年 12 月に Ruby 2.1 としてリリースした。さらにインクリメンタル GC アルゴリズムを実装した Ruby 処理系を 2014 年 12 月にリリース予定である。

本稿では、このライトバリアのないオブジェクトを考慮した世代別およびインクリメンタル GC について紹介し、Ruby への実装について述べる。

2. 世代別 GC とインクリメンタル GC

本章では世代別 GC とインクリメンタル GC についての概要を述べる。これらのアルゴリズムには、多くの基礎研究があるため、詳細は適切な文献を参照されたい。

2.1 世代別 GC

処理性能を上げるための手法の 1 つとして、世代別 GC[1] が知られている。世代別 GC は、オブジェクトの空間を、オブジェクトの世代ごとに複数に分割し、GC を行なう範囲を、可能な限り新しい世代のみに限定する手法である。若い世代のオブジェクトのみを対象とする GC をマイナー GC、古い世代のオブジェクトも対象にする GC をメジャー GC という。経験的に、マイナー GC のみで十分なことが多いため、この方法によって処理性能が改善することが知られている。

世代別 GC は古い世代から新しい世代への参照を発見し、間違って回収することがないように覚えておく（リメンバーセットに登録する）など、適切に処理する必要がある。マイナー GC では、リメンバーセットに登録されたオブジェクトをルートとすることで、正しく生きているオブジェクトとして扱うことができる。この世代間の参照を発見するために、ライトバリアというテクニックがよく利用される。

ライトバリアは、書き込み（ライト）により、あるオブジェクトの状態変更を検知するプログラミングテクニックである。世代別 GC では、ライトバリアによって、あるオブジェクト A から他のオブジェクト B へ参照が追加されたことを検知し、A と B の世代を確認し、A が B よりも古い世代であれば、リメンバーセットへ登録する。

例えば、Ruby プログラムでは、配列オブジェクト `ary` が古い世代のオブジェクトであり、オブジェクト `obj` が新しい世代のオブジェクトであったとき、`ary[0] = obj` というプログラムによって、旧世代のオブジェクトから新世代のオブジェクトへの参照が生まれる。このような操作を、ライトバリアによって検出する必要がある。

世代別 GC により、処理性能が改善する。ただし、マイナー GC による停止時間は、メジャー GC よりも短いが、メジャー GC による停止時間は長く、最悪停止時間は改善しない。

2.2 インクリメンタル GC

停止時間に関しては、短くするための手法の 1 つとして、インクリメンタル GC[1] が知られている。インクリメンタル GC は、GC 処理を複数の処理に分割し、対象プログラム（ミューテータ）と交互に進めていき、対象プログラム

の処理を長時間停止しないようにするテクニックである。ここでは、その中の 1 つを紹介する。

このアルゴリズムでは、オブジェクトに白、灰、黒という色をつけ、次のように進めていく。

(1) GC 開始時にすべてのオブジェクトを白色にする。
(2) ルートセットに含まれるオブジェクトを灰色にする。
(3) 灰色のオブジェクトのどれか（オブジェクト A）を取り出し、そこから参照されるオブジェクトを灰色にする。A から参照されるオブジェクトがすべて灰色、もしくは黒色とすれば、A を黒色にする。これを、灰色のオブジェクトがなくなるまで繰り返す。
(4) 白色のオブジェクトは、参照されないオブジェクトなので回収できる。

3 の手続きを分割し、ミューテータと交互に実行することで、対象プログラムの停止時間を短くする。ただし、対象プログラムの総実行時間が短くなるわけではない。

インクリメンタル GC では、ミューテータ実行時に黒色のオブジェクトから白色のオブジェクトへの参照が生成されることがある。このとき、黒色のオブジェクトから参照されるオブジェクトが灰色から黒色である、という前提条件が崩れてしまうため問題となる。

この黒色から白色のオブジェクトへの参照生成を検知するため、世代別 GC と同様にライトバリアを利用する。ライトバリアによって、オブジェクト A から B への参照が生成されたとき、A が黒色で B が白色だったとき、A を黒色から灰色に戻す、もしくは B を白色から灰色にする。

2.3 世代別インクリメンタル GC

世代別 GC とインクリメンタル GC は、とくに問題無く共存できるため、素直に世代別インクリメンタル GC を実装することができる。たとえば、停止時間が長いメジャー GC のみをインクリメンタルにすることがでる。

3. ライトバリアで保護しないオブジェクトを考慮した世代別インクリメンタル GC

1 章で説明したとおり、Ruby 処理系には適切なライトバリアが存在せず、従来の世代別 GC、もしくはインクリメンタル GC を導入するのは困難である。

ライトバリアを適切に挿入するには、(1) 大きな開発コストという問題と、(2) 互換性の問題がある。ライトバリアが必要な箇所で、1 箇所でも挿入を忘れると、重大な問題となるため、既存のソフトウェアを精査し、改変する必要がある。しかし、Ruby 処理系では、ライトバリアを必要とする箇所が多くあり、漏れなく対応するのは困難な作業である。また、Ruby 処理系を拡張するための拡張ライブラリにもライトバリアを必須とすると、過去に開発された拡張ライブラリが利用できなくなり、過去のソフトウェア資産を放棄することになり、プログラミング言語環境の魅力を大きく損ねることになる。

そこで、我々はライトバリアで保護しないオブジェクトを考慮した世代別 GC、およびインクリメンタル GC アルゴリズムを考案し、これを組み合わせ、ライトバリアが不完全であっても動作する世代別インクリメンタル GC を実現した。それぞれ制限付き世代別 GC（RGenGC: Restricted Generational GC）、および制限付きインクリメンタル GC（RincGC: Restricted Incremental GC）と呼ぶ。

3.1 キーアイデア

本提案のキーアイデアは、各オブジェクトに、ライトバリアが適切に処理されるかどうか区別する、というものである。あるオブジェクト A へ書き込みが行なわれ、A が他のオブジェクトへの参照があったことを検知することが保障できるとき、A をライトバリアで保護されたオブジェクト（以降、Protected オブジェクト）とし、そうでなければ A をライトバリアで保護されないオブジェクト（以降、Unprotected オブジェクト）とする。

例えば Ruby 処理系であれば、オブジェクトの取り扱いはクラスごとに実装されているため、あるクラス C の実装にライトバリアを完全に挿入すれば、クラス C のインスタンスはすべて、Protected オブジェクトといえる。それ以外の、未対応のクラスのインスタンスについては、Unprotected オブジェクトとする。Ruby における String クラスのオブジェクトを例にすると、ほとんど他のオブジェクトへの参照を生成しない [*2] ため、対応は容易である。クラスによって、この容易さは異なる。

また、Ruby 処理系の拡張ライブラリでは、ある操作を行なうと、それ以降あるオブジェクトについてライトバリアを保障することができないような操作が存在する。例えば、拡張ライブラリでは、Array（配列）クラスのオブジェクトの配列の要素を管理するメモリブロックのポインタを取得することができる（`RARRAY_PTR()` マクロ）。Ruby 処理系では、このポインタが指し示すメモリブロックにオブジェクトへの参照を書き込む、ということが許されており、拡張ライブラリでもこのようなプログラムはいくつか存在した。このような処理が行なわれると、参照の生成は検知できない。そこで、このようにある Array オブジェクトのメモリブロックのポインタを取り出したとき、Array オブジェクトが Protected オブジェクトであったなら、これを Unprotected オブジェクトへ変更する。これを、Unprotect 操作と呼ぶ。

まとめると、オブジェクトをライトバリアで保護する Protected オブジェクト、保護しない Unprotected オブジェ

[*2] Ruby のすべてのオブジェクトは、インスタンス変数経由でその他のオブジェクトを参照可能であるため、String オブジェクトが他のオブジェクトを参照することもある。ただし、このインスタンス変数の追加のような処理は、共通ルーチンを利用しているため、ライトバリアの挿入は容易である。

クトに大別し、ライトバリアで保護できないと判断できるタイミングで Protected オブジェクトを Unprotected オブジェクトへ変更する Unprotected 操作がある。

このアイデアを使い、世代別 GC、およびインクリメンタル GC アルゴリズムを実現する。

3.2 RGenGC: 制限付き世代別 GC

簡単のために、新世代と旧世代、2 世代を管理する世代別 GC を用いて説明する。

世代別 GC でライトバリアが必要となるのは、旧世代オブジェクトから新世代オブジェクトへの参照が生成されたことを検知するためである。ただし、Unprotected オブジェクトはライトバリアがないため、この生成を検知できない。

そこで、世代別 GC のアルゴリズムに次のルールを新たに加え、健全性を保つ。

(1) Unprotected オブジェクトは旧世代にしない。

(2) 旧世代のオブジェクトから Unprotected オブジェクト A への参照を発見したら、A をリメンバーセットに追加する。なお、リメンバーセットに追加された Unprotected オブジェクトは、次のメジャー GC までリメンバーセットから外されず、次のメジャー GC まで生き続ける [*3]。

(3) 旧世代オブジェクト B が Unprotect 操作により、Unprotected オブジェクトとなったら、B を新世代の Unprotected オブジェクトに変更し、B をリメンバーセットに追加する。

まず、(1) は、Unprotected オブジェクトは旧世代から新世代への参照を検知できないため、旧世代にすることはできないためである。

次に (2) は、旧世代オブジェクト A から参照される新世代オブジェクト B を発見したとき、B を旧世代にするなどして、適切に処理するが、B が Unprotected オブジェクトのとき、旧世代にすることができないため、リメンバーセットに登録し、マイナー GC 期間中、ずっと生きておくようにする。

(3) は、旧世代オブジェクト B が、他の旧世代オブジェクトのみから参照されている可能性があり、その時マイナー GC では誰からも辿ることができないため、B を生き残らせる必要があるためである。

この (1)〜(3) のルールを追加することで、世代別 GC を適切に機能させることができる。

なお、旧世代オブジェクトから参照される、リメンバーセットに追加された Unprotected オブジェクトを旧世代オブジェクトと呼ぶこともできるが、ライトバリア時に「旧世代オブジェクトから参照される新世代オブジェクト」を検知するが、リメンバーセットに追加された Unprotected オブジェクトは必ずルートセットとなるため、この検知対象とする必要がない。そのため、Unprotected オブジェクトは新世代オブジェクト、ということにする。

この方式では、リメンバーセットに登録された Unprotected オブジェクトがルートとして加わるため、従来の世代別 GC よりもマイナー GC にかかる時間が多くなる。

3.3 RincGC: 制限付きインクリメンタル GC

RGenGC と同様に、次のルールをインクリメンタル GC アルゴリズムに追加する。

(1) すべてのオブジェクトを黒色に変更した後、生きている黒色の Unprotected オブジェクトをルートセットとして、再度停止 GC を行なう。

黒色 Unprotected オブジェクトはライトバリアで保護されないため、ミューテータが実行中に白色のオブジェクトへの参照が生成されている可能性がある。そのため、インクリメンタル GC の最後に、全ての Unprotected オブジェクトをルートとした停止 GC を行い、生きている Unprotected オブジェクトから辿ることができるすべてのオブジェクトを黒色にする。

この 1 つのルールにより、インクリメンタル GC が適切に動作する。

本方式では、当然のことながら、生きている Unprotected オブジェクトの数に比例した停止時間がかかるため、従来のインクリメンタル GC よりも最悪停止時間が長くなる。

なお、インクリメンタル GC アルゴリズムの説明で、最後にルートの再スキャンする、としているものもある。これは、ミューテータによって、ルートセットに白色オブジェクトが加わる可能性があるためである。ルートを 1 つのオブジェクト R として捉え、R が Unprotected オブジェクトとすれば、本提案と同様となる。

3.4 議論

今回提案する RGenGC、および RincGC について議論する。

3.4.1 段階的な開発

従来、ライトバリアを必要とする GC アルゴリズムを導入するためには、ライトバリアが適切に実装されている必要があった。しかし、本提案では、まずはすべてを Unprotected オブジェクトとしておき、順次 Protected オブジェクトを増やす、という戦略をとることができる。この戦略では、まずは実装が容易な箇所や、よく利用されるオブジェクトからライトバリアを整備していくことができる。

例えば Ruby では、文字列オブジェクトや配列オブジェ

[*3] 旧世代オブジェクト X が新世代オブジェクト Y を参照したとき、X をリメンバーセットに入れるが、マイナー GC 終了後、リメンバーセットから外され、次のマイナー GC 時にはルートにならない。

クト、ハッシュオブジェクトが多く利用されるため、これら頻出オブジェクトを、まずは集中的に実装することができた。逆に、あまり利用されないような機能や、生成されるオブジェクトの数が少ないオブジェクトは、Unprotected オブジェクトのままでも処理性能、および停止時間に与える影響が少ないため、Protected オブジェクトに作り替えることを後回しにすることができ、開発リソースを有効に活用することができた。

また、本質的にライトバリアを入れるのが困難である、もしくはとても複雑なプログラミングが必要であるような場合、性能や停止時間を犠牲にしても、バグの無い実装を提供するため、Unprotected オブジェクトのままにしておく、という判断ができる。この点も、他のライトバリアを必要とする GC アルゴリズムには無い利点である。

例えば、Ruby 処理系においては、Module オブジェクトへの適切なライトバリアの挿入が困難であった。というのも、複数の Module オブジェクトが 1 つのデータ構造を共有していたためである。そのため、Module オブジェクトについてはライトバリア挿入を諦める、という判断を行なった。

3.4.2 RGenGC における性能低下

RGenGC では、リメンバーセットに登録される Unprotected オブジェクトが増えると、ルートセットが大きくなるため、従来の世代別 GC にくらべ、マイナー GC にかかる時間が長くなる。リメンバーセットに追加しなければならない Unprotected オブジェクトは、次の 2 通り考えられる。

(1) GC 時、旧世代オブジェクトから参照された Unprotected オブジェクト
(2) Unprotect 操作で旧世代オブジェクトから Unprotected オブジェクトとなったオブジェクト

経験上、旧世代オブジェクトからの参照はあまり増えることはないため、(1) は少ない。また、Unprotect 操作自体も、あまり多くないため、これらの条件での性能低下はあまり起きないので無いかと期待できる。

3.4.3 RincGC における停止時間の増加

RincGC では、インクリメンタル GC の最後に停止 GC によって、生きている（黒色の）Unprotected オブジェクトをルートセットとして再スキャンを行なう。そのため、少なくとも、生きている Unprotected オブジェクトの数に比例した停止時間がかかる。

RGenGC のマイナー GC の停止時間を考えてみると、すべての生きている Unprotected オブジェクトを辿るため、少なくとも、この数に比例した停止時間がかかる。つまり、マイナー GC の停止時間と同等か、それより短いことが期待できる。

3.4.4 ライトバリア挿入漏れ

本手法でも、あるオブジェクトを Protected オブジェクトとしていながら、ライトバリアの実装漏れが存在すると、重大な問題となる。ライトバリアの挿入漏れを防ぐためには、別途チェック機構を用意する必要がある。

4. Ruby への実装

我々は、提案する手法を Ruby 処理系に構築した。本章では、実装のポイントとなる部分を紹介する。

4.1 世代別インクリメンタルマーキング

今回は、保守的マークアンドスイープ GC のマーク部分を世代別・インクリメンタル GC とした。もともと、スイープ処理をインクリメンタルに処理する Lazy sweep という機構は実装されていたため、マーク部をインクリメンタル GC とした。

保守的であるため、マイナー GC においてよく利用されるコピー GC などの移動型 GC アルゴリズムは利用できないため、マイナー GC、およびメジャー GC ともにマークアンドスイープ GC を用いている。

世代は新世代と旧世代の 2 世代として実装した。Protected オブジェクトは、4 回 GC を生き残ると、旧世代オブジェクトに昇格するようにした。

4.2 ビットマップの利用

もともと、Ruby 処理系ではあるオブジェクトがマークされたことを表現するために、各オブジェクトに対応する 1 ビットを集めた、ビットマップとして表現していた（Mark bitmap）。そこで、これに加える形で次のビットマップを用意した。

(1) Unprotected bitmap
(2) Long lived bitmap
(3) Marking bitmap
(4) Remember old bitmap

Unprotected bitmap では、オブジェクトが Protected か Unprotected かを示す。Long lived bitmap では、世代別 GC における旧世代オブジェクトか、リメンバーセットに登録された Unprotected オブジェクトが表現される。Marking bitmap は、インクリメンタル GC における灰色オブジェクトを表現する。Remembered old bitmap では、世代別 GC における、リメンバーセットに登録された旧世代オブジェクトを表現する。

例えば、マイナー GC 時にマークする必要があるリメンバーセットは、次のように取得することができる。Remembered old bitmap と（Long lived bitmap と Unprotected bitmap の論理積）の論理和で得ることができる。

また、RincGC で最後に必要となる生きている Unprotected オブジェクトの集合は、Mark bitmap と Unprotected bitmap の論理積で得ることができる。

なお、Remember old bitmap はマイナー GC でのみ、

Marking bitmap はメジャー GC でのみ利用されるため、同じメモリ領域を共用している。

RGenGC および RincGC のために、各オブジェクトに 3 ビット追加していることになる。100 万オブジェクト（1M オブジェクト）で試算すると、$3bit*1M = 3Mbit = 375KB$ のメモリ使用量の増加となる。

4.3 RGenGC におけるメジャー GC タイミング

Ruby 処理系は GC 終了時、確保できる空きスロットの数が少なければ、ヒープを拡張する。この時、マイナー GC 時に空きスロットの数が少なければ、メジャー GC を実行する、というようにした。

5. 評価

Ruby 処理系に実装した RGenGC、および、それに RincGC を加えたもので本手法の評価を行なう。評価環境は Intel(R) Core(TM) i7-4930K CPU @ 3.40GHz 上の Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-37-generic x86_64) で行なった。対象とする Ruby 処理系は ruby 2.2.0dev (2014-11-09 trunk 48334) [x86_64-linux] を用いた。

5.1 RGenGC

まず、次に示すマイクロベンチマークを実行し、マーク時間、およびスイープ時間がどのようになるか調査する。スイープの時間計測を確実にするため、インクリメンタルにスイープを行なう Lazy Sweep 機構は切って評価を行なった。

```
# Make 1M objects (long lived)
ary = (1..1_000_000).map{|e| Object.new}
# Make short-lived 100M objects
100_000_000.times{Object.new}
```

このプログラムでは、まず旧世代のオブジェクトを 100 万個作成し、さらに短寿命オブジェクトを 1 億個生成するというプログラムである。結果を図 1、図 2 に示す。

図 1 を見ると、各 GC において、RGenGC によって、マーク時間が大幅に短縮できていることがわかる。ただし、スイープ時間は余分にかかっていることが確認できる。スイープ時間には変更がないと期待されるため、何らかの実装の不備が考えられる。

次に、Rdoc という Ruby のソースコードを読み込み、ドキュメントを生成するプログラムを走らせた結果を、図 3、図 4、図 5 に示す。

図 3 を見ると、マイナー GC により、ほとんどの GC 時間が少なくなっていることがわかる。また、いくつかのピークが観測できるが、これはメジャー GC であると予想される。結果的に、マークにかかる時間を 8 倍程度圧縮することができたことがわかった。

図 6 に、RDoc プログラムを実行されたとき、確保されたスロット（1 オブジェクトを格納する領域）の数、生きているオブジェクトを格納しているスロットの数（つまり、生きているオブジェクトの数）、旧世代のオブジェクトの数、Unprotected オブジェクトの数、リメンバーセットに記録された Unprotected オブジェクトの数がどのように推移するかを示す。また、図 7 に後者 2 つのみの結果を表示する。これを見ると、RDoc プログラムでは、Unprotected オブジェクト、およびリメンバーセットに記録された Unprotected オブジェクトの数が、他のオブジェクトに比べて十分少ないことが確認できる。そのため、マイナー GC への性能の影響が殆ど出なかったと考えられる。また、Unprotected オブジェクトが増えても、リメンバーセットに記録されるものは少ない、ということがわかる。

5.2 RincGC

図 8 に、前節で示したマクロベンチマークプログラムの、GC による停止時間の長い順に上位 2048 個を、RGenGC（ただし、強制的にメジャー GC を起動する）、および RincGC で実行した結果を示す。

結果を見ると、意図と反して RincGC のほうが最悪停止時間が長いことがわかる。まだ詳しく調査出来ていないが、実装上の問題ではないかと思われる。

RDoc アプリケーションでは、通常の RGenGC、および、それに RincGC を加えたもので実行し、GC 処理による停止時間の長い順に上位 256 個を表示したものを図 9 に示す。こちらでは若干効果が見えるが、あまり大きな効果ではない。インクリメンタル GC の起動タイミングや各ステップの停止時間の選択方法などに、まだ課題が残っていると思われる。

6. 関連研究

世代別 GC、およびインクリメンタル GC には関連研究が多くあるが、必要となるライトバリアをいかに挿入するか、という点を議論する研究は多くない。これは、モダンなプログラミング言語処理系では、ライトバリア（もしくはそれに代わるもの）を前提に開発が進められるからではないかと思われる。

数少ない研究として、花井らは処理系を記述する言語を解析し、自動的にライトバリアを挿入する手法を提案している [2]。この手法では、ほぼ漏れなくライトバリアが挿入できるが、解析の限界による挿入漏れがあり得ること、挿入のしすぎがあり得ること、それから専用の解析器が必要となることといった問題点がある。

ライトバリアではなく、ハードウェアのページ保護機能を用いてライトバリアの代わりにする手法もよく知られている [1] が、ページ保護機能は移植性が低いという点や、Ruby のデータ構造にあわない、という問題がある。

7. まとめ

本稿では、正確なライトバリアが必須であるとされる世代別 GC、およびインクリメンタル GC について、ライトバリアを適切に導入することが困難な Ruby 処理系に導入するために、ライトバリアに対応した Protected オブジェクトと対応しない Unprotected オブジェクトを用いて、世代別 GC、およびインクリメンタル GC を実現するアルゴリズムを提案した。そして、提案したアルゴリズムを Ruby 処理系に実装し、性能改善、および停止時間の短縮に効果があることを示した。

今回実装した世代別 GC では、マーク処理のみを世代別とするため、スイープ処理の時間は変わらない。そのため、スイープ時間のさらなる削減が必要である。

開発した Ruby 処理系は、RGenGC を組み込んだものを Ruby 2.1 として 2013 年 12 月にリリースし、これに RincGC を組み込んだものを Ruby 2.2 として 2014 年 12 月にリリースする予定である。

参考文献

[1] Jones, R., Hosking, A. and Moss, E.: *The Garbage Collection Handbook: The Art of Automatic Memory Management*, CRC Applied Algorithms and Data Structures, Chapman & Hall (2012).

[2] 花井亮，小宮常康，八杉昌宏，湯淺太一：Scheme 処理系における C 言語拡張コードへのライトバリア自動挿入，情報処理学会論文誌. プログラミング， Vol. 44, No. 4, pp. 17–24（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/110002711929/⟩ (2003).

[3] 松本行弘：Ruby の真実，情報処理， Vol. 44, No. 5, pp. 515–521 (2003).

[4] 笹田耕一，松本行弘，前田敦司，並木美太郎：Ruby 用仮想マシン YARV の実装と評価，情報処理学会論文誌 (PRO), Vol. 47, No. SIG 2(PRO28), pp. 57–73 (2006).

図1. RGenGC: Micro-benchmark

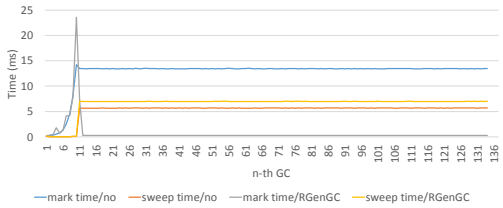

図2. RGenGC: Micro-benchmark

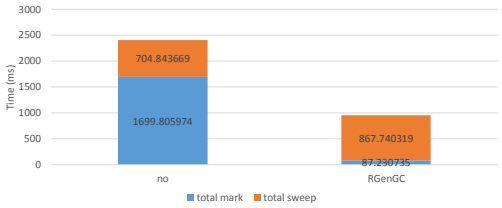

図3. RGenGC: Rdoc application

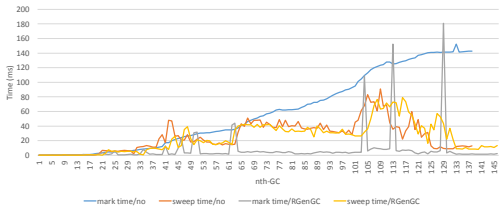

図4. RGenGC: Rdoc application

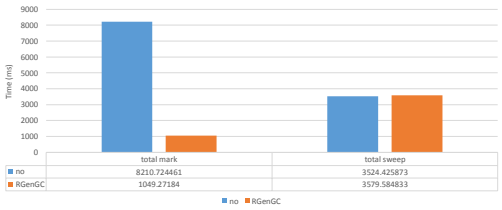

図5. RGenGC: Rdoc application

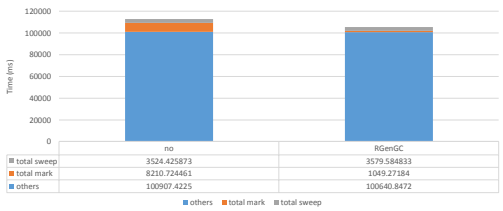

図6. RGenGC: Rdoc application

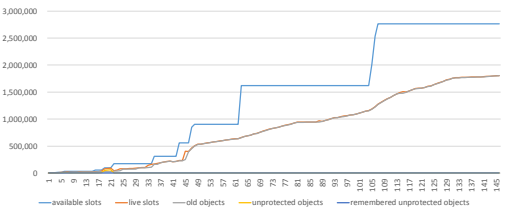

図7. RGenGC: Rdoc application

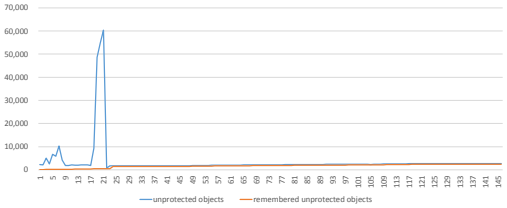

図8. RincGC: Micro-benchmark

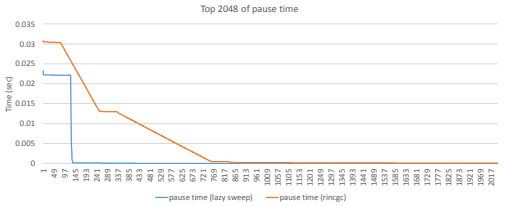

図9. RincGC: Rdoc application