システムＬＳＩ設計技術　109－8
(2003. 3. 11)

スナップショット方式によるJavaのGCのリアルタイム化

遠藤匠† 田中陽†† 前田敦司††† 山口喜教†††

† 筑波大学　第三学群情報学類　〒305-8577　茨城県つくば市天王台1-1-1
†† 川崎マイクロエレクトロニクス株式会社　〒261-8501　千葉市美浜区中瀬1-3
††† 筑波大学　電子・情報工学系　〒305-8577　茨城県つくば市天王台1-1-1
E-mail: †endou@ialab.is.tsukuba.ac.jp, †††{maeda,yamaguti}@is.tsukuba.ac.jp

あらまし　近年、Javaはその利用範囲の拡大に伴い、処理系の性能の大幅な向上が求められている。リアルタイム化もそのひとつである。しかし、一般的なJava処理系はリアルタイム性がない。主な原因のひとつとして、GCによるユーザープログラムの不確定な停止時間がある。本研究では、スナップショットと呼ばれる手法を用いたGCのインクリメンタル化によって、そのリアルタイム化を行った。また、インクリメンタル化に伴う処理系の誤動作の問題についても、ライトバリア、リターンバリアと呼ばれる手法を用いて解決をした。

結果として、GCによるユーザープログラムの最大停止時間を大きく減少させ、特定のシステムに対してそのリアルタイム性を保証することができるようになった。

キーワード　実行時システム、メモリ管理、リアルタイム

A Real-Time Garbage Collection for Java using Snapshot Algorithm

Takumi ENDO†, Yo TANAKA††, Atsushi MAEDA†††, and Yoshinori YAMAGUCHI†††

† College of Information Sciences, University of Tsukuba, 1-1-1 Tennoudai, Tsukuba 305-8577 JAPAN
†† KAWASAKI MICROELECTRONICS, INC., 1-3, Nakase, Mihama-ku, Chiba 261-8501, JAPAN
††† Institute of Information Sciences and Electronics, University of Tsukuba, 1-1-1 Tennoudai, Tsukuba 305-8577 JAPAN
E-mail: †endou@ialab.is.tsukuba.ac.jp, †††{maeda,yamaguti}@is.tsukuba.ac.jp

Abstract　Recently, as the use of Java spreads, performance requirements for Java implementations are increasing. Real-time processing ability is one such requirement for Java implementations. However, common Java impementations cannot meet real-time requirements mainly because unbounded pause time of garbage collector (GC). In this report, we present a real-time garbage collector based on Yuasa's Snapsho-at-the-Beginning incremental collection algorithm. We have implemented write-barrier and return-barrier to compensate pointer modification by mutator.

As a result, we could drastically reduce the maximum pause time of the user program caused by GC and have successfully guaranteed the real-time processing of specific kind of applications.

Key words　runtime system, memory management, real-time system

1. 序　　論

Javaやその他の多くの言語処理系では、プログラマの動的メモリ管理に関する誤りを大幅に減らし、生産性、信頼性を向上させる自動記憶管理の機能をガーベジコレクション（以下GC）によって実現している。しかし一方で、このGCの機能がJavaによるリアルタイム処理を困難にしている原因の一つとなっている。なぜならば、通常のJava処理系においては、GCによるユーザープログラムの最大停止時間が不確定であり、プログラムがリアルタイム性を得ることができないからである。

本研究では、このGCによるユーザープログラムの最大停止時間に上限を設け、それを保証することでGCのリアルタイム化を試みる。

なお、実装にはOmni OpenMP Compiler[4]に同梱されたJava処理系jexcを用いる。jexcは、Javaバイトコードから Cのコードへのトランスレータである。

2. マーク・スイープGC

代表的なGCのアルゴリズムとしては、参照カウンタ法、マーク・スイープGC、コピーGCなど[5]がある。ここでは、

jexc で用いられているマーク・スイープ GC について説明する。

マーク・スイープ GC は、ルートが保持するポインタから、たどれるオブジェクトが保持するポインタの先を再帰的に探索し、到達可能なオブジェクトには生きているオブジェクトとして印を付け（マークフェーズ）、印のないオブジェクトをごみとして回収し、再び利用可能な自由セルとしてフリーリストに繋いでいく（スイープフェーズ）方法である。この方法には、以下のような欠点がある。

（1） GC によるユーザープログラムの停止時間が不確定であり、リアルタイム性が失われる。

（2） フリーリストには、ヒープ中に断片的に存在する様々なサイズの自由セルがつながれているため、自由セルの総量にかかわらず、割り当て要求に適したサイズの自由セルが確保できない可能性がある（フラグメンテーション問題）。

本研究では、マーク・スイープ GC における（1）の問題点を、スナップショットと呼ばれる手法を用いて解決し、そのリアルタイム化を行う。

3. スナップショット GC

マーク・スイープ GC は、マークフェーズによるユーザープログラムの停止時間が探索するオブジェクトの数に依存し、その上限が保証されないために、リアルタイム性がない。

また jexc において、ヒープは CHUNK と呼ばれる 32M バイトの領域のリストからなっている。GC を行っても要求を満たすサイズの自由セルが存在しないとき、新たに CHUNK を確保してヒープを拡大する。したがってヒープ全体のサイズはプログラム中に動的に変化するため、ヒープのサイズに依存するスイープフェーズの最大停止時間も、その上限が保証されない。

GC のリアルタイム化を実現するには、GC によるユーザープログラムの停止時間に上限を設けてやればよい。本研究においては、ユーザープログラムの中に GC 作業を時間的に分散させることで、その最大停止時間に上限を設ける。これをインクリメンタル化と呼ぶ。インクリメンタル GC においては、GC 処理にある決められた作業量を設定して、その作業量の処理が終了したらユーザープログラムに処理を返してやるようにすれば、GC によるユーザープログラムの停止時間に上限を設けることができる。

インクリメンタル化の手法として、本研究ではスナップショットと呼ばれる手法を用いる。スナップショット GC [2] では、ルートとなるポインタを GC 処理用のスタックに全てコピー（スナップショットを取る）して、そこから参照可能な全てのオブジェクトに印を付けている。スナップショットを取った後に生成されたオブジェクトについてはマーク処理は行われず、次回の GC の際にごみかどうか判別する対象となる。

4. GC のリアルタイム化：rt_jexc

jexc を改良し、そのメモリアロケーションと GC のリアルタイム化を行った rt_jexc について説明する。

4.1 メモリアロケーションのリアルタイム化 [1]

jexc のメモリアロケータにはリアルタイム性がない。その原因は、アロケータ中の自由セルの線形探索である。そこで rt_jexc では、フリーリストの構造に変更を加え、検索に制限を与えることでより高速かつリアルタイムなメモリアロケーションを実現した。

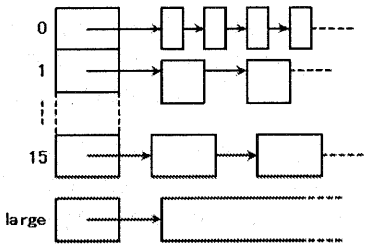

図 1 jexc のフリーリストの構造

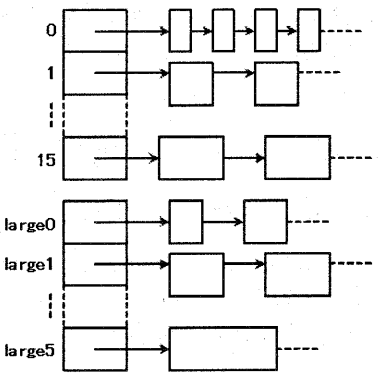

図 2 rt_jexc のフリーリストの構造

jexc では、オブジェクト生成のために要求されるサイズ、そしてヒープ中のセルのサイズは、ともに 16 バイト単位である。16〜256 バイトの自由セルは、小領域専用の配列にフリーリストとして繋がれている。256 バイト以上の自由セルは、すべて大領域用のひとつのフリーリストに繋がれている。（図 1）rt_jexc では 256 バイト以上の自由セルについても、そのサイズごとに専用の配列を用意した。i 番目の配列には、$128*8^i$〜$128*8^{i+1}$ のサイズの自由セルが繋がれている。（図 2）

アロケータ側では、自由セルの参照回数に上限を設定し、それを超えたら探索を中止し、CHUNK を確保してそれを要求に充てるようにする。

以上の改良によって、アロケータの高速化・リアルタイム化が実現する。

4.2 マークフェーズ・スイープフェーズのリアルタイム化 [1]

GC のリアルタイム化を行うためには、マークフェーズ、スイープフェーズをそれぞれインクリメンタル化する必要がある。GC は基本的にはアロケータの中で呼び出される。ユーザープログラムからセルの割り当て要求がある度に、GC の作業を少しずつ、インクリメンタルに進めていく。インクリメンタルに

進めていく単位は、マークの際に参照するオブジェクトの数や、オブジェクトを回収するためにたどったりするセルの数として決めてやればよい。

インクリメンタル化したGCでは、GC中に定期的にユーザープログラムに処理を返すことになるため、この間に生成されるオブジェクトの処理に対処する必要が生じる。新たに生成されるオブジェクトは、生成された瞬間はどこからも参照されてはいない。いつかは参照されるようになるかもしれないが、いつ参照されるようになるかは、処理系が預かり知らぬところである。しかし、GC自体にはこれをマークする手立てはない。したがって、このオブジェクトはアロケータの責任において必ずマークする必要がある。

4.3 ライトバリア

マークフェーズのインクリメンタル化によって、マークフェーズ中にもオブジェクト間のポインタの書き換えが発生するようになる。すると、図3のような破壊的な書き換えが起こり、生きているはずのセルがマークされなくなる可能性がある。

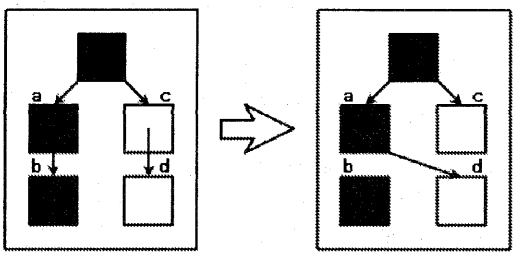

図3 ポインタの破壊的な書き換え

マークフェーズによるマーク処理が図3左のようにbまで進んだ状態で中断したとする。その後、ユーザープログラムによってポインタ変数の書き換えが起こり、図3右のように、aのポインタ変数がdへのポインタに書き換えられ、cのポインタ変数がdへのポインタではなくなったとする。この状態で再びマークフェーズが再開されたとすると、マーク処理はcから始まり、dがマークされなくなってしまう。結果として、生きているオブジェクトであるはずのdはGCによって回収されてしまう。

このような、オブジェクト間のポインタの破壊的な書き換えによるマークミスを防ぐには、ポインタ変数の書き換えが起こるとき、それによって参照されなくなるオブジェクトへのポインタをGC用のスタックに積んでやればよい。図3の例では、オブジェクトdがオブジェクトcから参照されなくなるときdをスタックに積む。この操作は、GC処理プロセスではなく、ユーザープログラム処理プロセスの責任において行う。このように、書込み操作の度にチェックを行う手法をライトバリア[2]と呼ぶ。

4.4 ルートコピーのインクリメンタル化

スナップショット方式のGCにおいてはルートとなるポインタをGC処理用のスタックにコピーする処理をアトミックに行わなければならない。そのため、コピー処理によるユーザープログラムの停止時間がルートのサイズに依存し、GCのリアルタイム化の障害となる。そこで、ルートコピーの処理をマークフェーズ、スイープフェーズと同様にインクリメンタル化することで、リアルタイム化を行う。ルートをコピーした個数が決められた一定回数に達した時、処理を中断してユーザープログラムに処理を返す。その際、最後に参照していた個所を覚えておき、再びGCのプロセスに入ったときに、その個所からコピー処理を再開できるようにする。

jexcにおいて、ルートはルートスタック、クラステーブル、Stringハッシュテーブルの3つの領域からなる。そのうち、インクリメンタル化の対象となるのはサイズが動的に変化するルートスタックである。ルートスタックはユーザープログラムがプログラムを実行するのに用いる領域であり、関数フレームがPUSH、POPされる。関数の実行中は、現在実行中の関数のフレーム（カレントフレーム）内にしか読み書きを行わない。

また、マークフェーズ、スイープフェーズのインクリメンタル化のときと同様に、コピー中に生成されるオブジェクトに対処する必要がある。コピー中に新たに生成されるオブジェクトへのポインタは、ルートのトップ側へと積まれていく。ルートのコピーはトップからボトム側へと進めていくため、新たなオブジェクトへのポインタはGC用スタックにコピーされない。したがって、これらのオブジェクトはアロケータの責任において必ずマークする必要がある。

4.5 リターンバリア

ルートコピーをインクリメンタル化する際、忘れてはならない問題がある。マークフェーズのインクリメンタル化の際にオブジェクトへのポインタの破壊的な書き換えが起こることを述べたが、ルートコピーについても同様に、コピー中のポインタの書き換えによって生きているはずのセルへのポインタがコピーされない可能性がある。そのような例を図4に示す。

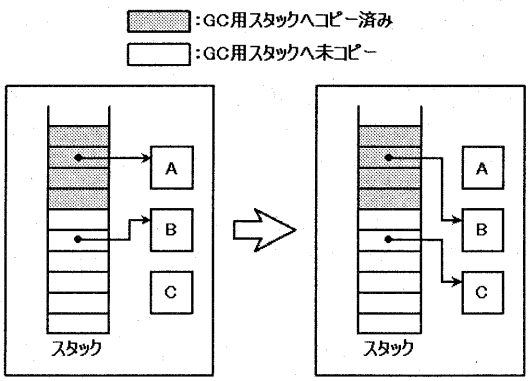

図4 インクリメンタル化における問題点

ルートコピーのインクリメンタル化によって、図4左の状態でGC処理の処理を中断し、ユーザープログラムに処理を返したとする。この状態では、セルAへのポインタはGC用スタッ

クにコピーされており、セルBへのポインタはまだコピーされていない。そして、ユーザープログラムによってスタック中のポインタが書き換えられ、図4右の状態になったとする。この状態では、セルBはルート中のポインタから指されているが、図4左でセルBを指していたポインタが別のセルCを指すように書き換えられている。この状態からGC処理が起動し、スタックのコピーを再開しても、セルBを指しているポインタはセルAを指すポインタとしてコピーされてしまっているので、セルBがマークフェーズでマークされることはない。するとセルBは、生きているセルであるにもかかわらず、ごみとして回収されてしまう。

このような問題が起こるのは、ルート中のまだコピーされていない領域で書き換えが起こってしまったのが原因である。この問題は、ライトバリアを用いて解決することができる。すなわち、Bを指していたポインタが書き換えられたとき、参照されなくなったセルBへのポインタをGC用スタックにコピーしてやればよい。しかし、プログラム中で頻繁に書き換えの起こるルートに対してライトバリアを適用するのは、そのオーバヘッドの大きさが無視できなくなり、現実的ではない。そこでrt_jexcでは、上述した「関数の実行中はカレントフレーム内にしか読み書きを行わない」という性質を利用し、この問題を以下のような手法を用いて解決する。

- スタック内のコピーされた部分とまだコピーされていない部分の境目をバリアと呼ぶ。
- カレントフレームがバリアよりも下位（ボトムの方向）に移ることを防ぐ。
- 関数からリターンする直前に、リターン後の関数フレームがすべてコピー済みであるかどうかをチェックする。
- リターン後のカレントフレームがまだすべてコピーされていない場合、ユーザープログラムの実行を中断し、リターン後の関数フレームをすべてGC用のスタックにPUSHする（図5）。

書き込み操作の度にチェックを行うライトバリアに対し、このように関数からのリターン時にチェックを行う手法をリターンバリア[3]と呼ぶ。

図5(1)の状態において、実行中の関数からリターンして5(2)になったとする。すると、カレントフレームがバリアよりも下位に移動してしまい、ポインタの破壊的な書き換えが起こる可能性が生じてしまう。

そこで、図5(3)のように実行中の関数からリターンするとカレントフレームがバリアを越えてしまうような場合、リターンする直前にユーザープログラムの処理を中断し、図5(4)のように、リターン後の関数フレームをすべてGC用スタックにコピーする。その後リターンすれば、図5(5)のようにカレントフレームがすべてコピー済みの状態となり、ポインタの破壊的な書き換えを防ぐことができる。

上述したリターンバリアをjexcのリアルタイム化に対して実装を行う際に、jexcにおけるインプリメンテーション上の制約がある。jexcでは、カレントフレームからはリターン後の関数フレームがどこから開始しているかを知ることができない。

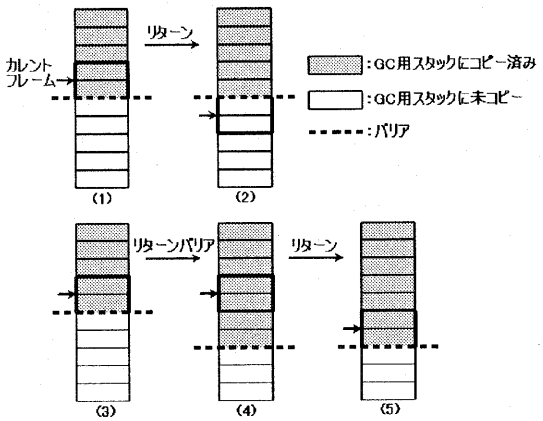

図5　リターンバリア

そのため、関数からリターンする直前に、リターン後の関数フレームがすべてコピー済みであるかどうかを判別できない。そのため、今回の実装ではこの判別をリターンする直前ではなく、リターンした直後に行うようにした。この方法では、リターンしてカレントフレームが移動した直後に、そのフレームがすべてコピー済みであるかどうかを調べ、コピーされていない箇所があればユーザープログラムの処理を中断し、その部分をコピーすることにより、同様の特性を実装した。

また、この実装方法を用いることによって、catch&throw機構のような非局所的脱出など、リターン後にカレントフレームが変則的に移動するようなプログラムでも、問題なく動作するという利点も生じた。

5. 評価と考察

rt_jexcの評価を行う。評価には、アッカーマン関数のプログラムを用いる。アッカーマン関数は、引数が増えるごとに、必要なヒープの大きさやオブジェクトの総数、ルートのサイズなどが指数関数的に大きくなるのが特徴である。

本研究における評価環境は次の通りである。

WorkStation	Sun ULTRA 10
CPU	ULTRA SPARC-IIi 333MHz
Memory	768MB
OS	Solaris 2.6
C Compiler	gcc 2.95.2

また、インクリメンタル化の単位として、メモリ割り当て要求のたびにコピーするルートの個数（COPY_STEP）を10、印付け処理を行うオブジェクトの参照個数（MARK_STEP）を5、メモリ割り当て要求のたびにごみを判別するセル参照個数（SWEEP_STEP）を5に設定する。

評価においては、マークフェーズやスイープフェーズを含むすべてのGCによるユーザープログラムの最大停止時間を測定

した。その結果を表 1 に示す。

	rt_jexc	jexc
ack(3,8)	415	156501
ack(3,9)	445	159060
ack(3,10)	501	168659
ack(3,11)	558	188732
ack(3,12)	598	249009

表 1　ユーザープログラムの最大停止時間 (μs)

rt_jexc では、マークフェーズ・スイープフェーズのインクリメンタル化や、ルートコピーのインクリメンタル化によって、jexc と比べて GC によるユーザープログラムの最大停止時間が大きく減少していることが分かる。加えて、jexc はアッカーマンの引数が増大するごとにその最大停止時間も大きく増加するのに対し、rt_jexc は最大停止時間がほんの少しずつ増えていくだけである。

しかしながら、rt_jexc においても、GC の完全なリアルタイム化は実現することができていない。インクリメンタル化を行った各フェーズごとに測定した場合においても、最大停止時間がほんの少しずつ増加する傾向にあり、その上限を完全に保証することはできなかった。これについては、現時点ではその原因を明らかにすることができていない。

しかし、完全なリアルタイム化こそ実現できなかったものの、COPY_STEP や MARK_STEP、SWEEP_STEP といったパラメータを調整することにより、特定のシステムに対してであれば、それが要求する最大停止時間を保証することができる。

加えて、ライトバリア、リターンバリアをそれぞれ実装したことにより、GC のリアルタイム性のみならず、その正常動作をも保証できたことになる。

最後に、jexc と rt_jexc における、アッカーマンの 5 種類の引数における総実行時間を比較する。測定結果を表 2 に、グラフを図 6 に示した。

	rt_jexc	jexc
ack(3,8)	4551	3132
ack(3,9)	18239	12275
ack(3,10)	76314	49642
ack(3,11)	338361	210630
ack(3,12)	1527882	942689

表 2　アッカーマンの総実行時間の比較 (ms)

rt_jexc の方が総実行時間が大きくなっているのが明らかである。その原因としてはインクリメンタル化によるプロセスの切り替えのオーバーヘッドや、ライトバリア、リターンバリアによるオーバーヘッドなどの他に、スナップショットのアルゴリズムでは、GC サイクル中に割り当てられたオブジェクトがすぐにごみになっても、そのサイクルでは回収できないということも大きな原因として考えられる。一般的に、ヒープサイズが大きいほど GC の回数が減り、生きているオブジェクトの量が

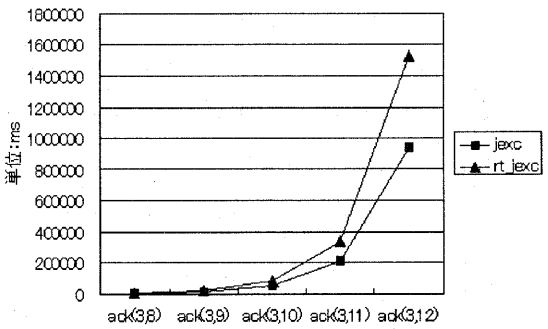

図 6　アッカーマンの総実行時間の比較（グラフ）

少ないほどマーキングに要する時間は減るため、空き領域の割合が大きいほど GC の効率は良くなる。しかしスナップショット GC では、一括型 GC に比べて「生きている」とみなしてしまうオブジェクトが多いため、空き領域の割合が少なくなってしまう。その結果、GC の効率を大きく低下させ、総実行時間が大きくなってしまう。実行時間の増加は 45-62%と、アッカーマンの引数が大きくなるにつれて、GC の処理回数やライトバリア・リターンバリアの頻度も増え、その差が大きくなっていくことがわかる。本研究では GC のリアルタイム化のみに注目していたために、総実行時間の増大は無視できない大きさとなった。リアルタイム性を損なわず、なおかつより高速な処理系の動作を実現するのが今後の課題である。

6. 結　　論

本研究では、スナップショットの手法を用いた GC の処理のインクリメンタル化によって、ユーザープログラムの最大停止時間の上限を保証することを目指し、その評価を行った。また、GC 同様にリアルタイム性のないメモリアロケーションの処理についてもリアルタイム化を行った。

マークフェーズ・スイープフェーズのインクリメンタル化に加え、ルートコピーにもインクリメンタル化を施したことで、GC 全体としての最大停止時間を大きく減少させることができた。しかし、問題の規模の増大に伴い、その最大停止時間もわずかながら増大してしまうという問題が生じ、完全なリアルタイム GC の実現には至らなかった。本研究ではその原因を明らかにできなかったが、ルートやヒープのサイズの拡大による、キャッシュミスやページフォルトなどが原因として考えられる。

マークフェーズのインクリメンタル化やルートコピーのインクリメンタル化によって、ポインタの破壊的な書き換えによるマークミスが生じる可能性があるが、ライトバリア、リターンバリアの手法を用いてそれを解決した。

今回解決できなかった最大停止時間の上限の完全な保証や、プログラム全体の処理速度、メモリ空間の有効利用などといった処理効率の向上などが、今後の課題として挙げられる。

謝　　辞

本研究の一部は科研費 (13680389) による補助を受けた。

文　献

[1] 田中陽. Java のガーベジコレクションのリアルタイム化に関する研究. 筑波大学第三学群情報学類平成 13 年度卒業研究論文. 2002

[2] 湯浅太一. 実時間ごみ集め. 情報処理, Vol.35, No.11, pp. 1006-1013, 1994

[3] 湯浅太一, 中川雄一郎, 小宮常康, 八杉昌宏. リターン・バリア. 情報処理学会論文誌, 第四一巻, No.SIG 9, pp. 87-99, 2000

[4] K.Kusano, S.Satoh, and M.Sato. Performance evaluation of the omni openmp compiler. In ISHPC 2000, pages 403-414, 2000.

[5] Richard Jones, Rafael Lins. Garbage Collection Garbage Algorithms for Automatic Dynamic Memory Management, 1996

[6] M.S.Johnstone and P.R.Wilson. The memory fragmentation problem: Solved? In International Symposium on Memory Management, British Columbia, Canada, Oct, 1998