記号処理 41－5
（1987. 6. 26）

実時間ＧＣの実装と評価

小沢 年弘　　林 耕司　　服部 彰
富士通研究所

人工知能用言語を実時間処理に適用する場合、ガーベジ・コレクション（ＧＣ）によるリスト処理の長時間の中断をさける必要がある。

本論文では、コピー法に基づく実時間ＧＣの改良方式を提案する。データアクセス時におけるオーバーヘッドを削減することにより、実行効率を高めることを目標にした。これは、データ構造を変更することなく、データアクセス時におけるポインタの状態の判定を簡素化することにより達成されている。

この方式をインプリメントし、従来の実時間ＧＣおよび一括型ＧＣとの性能比較を行っている。

An Implementaion and Evaluation of Real Time GC

Toshihiro OZAWA　Kohji HAYASHI　Akira HATTORI
Fujitsu Laboratories Ltd., 1015 Kamikodanaka Nakahara-ku, Kawasaki, 211, Japan

When we use an artifficial intelligence language in real-time applications, we need to solve that the garbage collection(GC) suspends the operation of a list processor for a long time.

In this paper, we propose the improved method of real-time GC which is based on copying one. We expect to reduce the overhead of the data access primitives to get higher execution efficiency. This is attained by simplifying the check of GC status of pointer in the data access, without changing data structures.

We implement and evaulate it in comparison to original copying real-time GC and stop-and-collect copying GC.

1．はじめに

人工知能をプロセス・コントロール等の実時間処理に応用する場合、ガーベジ・コレクション（ＧＣ）により、人工知能用言語の処理が長時間中断することが問題となる。また、このことが、プログラム開発環境としての人工知能用言語処理系のマンマシン・インタフェースを悪化させている。これらのことを解決するために、ＧＣプロセスとリスト・プロセスとを同時に働かす並列形ＧＣやＧＣプロセスをリスト・プロセスの中に分割する実時間形ＧＣなどが研究されてきた。しかし、ＧＣ専用ハードウェアのない汎用機においては、効率的な処理系が開発されていない。本稿では、汎用機上におけるLisp言語のコピー法に基づく実時間ＧＣの性能を評価するとともに、より効率的な実時間化を目指して、コピー法に基づく実時間ＧＣの改良方式を提案しその評価を述べる。

2．Lispのコピー法に基づく実時間ＧＣ

コピー法に基づく実時間ＧＣは、Baker により、一括型ＧＣであるコピー法ＧＣの修正として提案された(1)。コピー法ＧＣでは、記憶領域を二つの空間（新空間と旧空間）に分割し、リスト処理側では、新空間のみを使用する。現在使用中の新空間を使い果し、ＧＣが起動されると、新空間と旧空間を入換え（flip）、生きているデータを、手繰りながら新しい新空間にコピーする。このコピーは、次のような手順で行われる（図１参照）。つまり、レジスタＢは常に新空間の空き領域を指し、データはレジスタＢが指す領域に、順にコピーされる。コピーされたデータでその内容をまだ手繰っていないデータの先頭は、レジスタＳにより指されている。レジスタＳは、新空間をスキャンすることによって、次に手繰るべきデータを追っていく。レジスタＳが、レジスタＢに追い付いた時が、ＧＣが終了した時である。同一データに対する多重参照関係を保持するために、旧空間のコピー元データには、新空間のコピー先アドレスを記入しておく（このポインタを、forwarding pointerと呼ぶ。）。

Baker の方式でも新空間の未使用領域を使い尽すと、flipを起し、データのコピーを始める。しかし、実時間性を持たせるために、一度にコピーするデータの数を一定値Ｋ個以下に制限している。コピーは、領域獲得毎に行われる。つまり、ＧＣは、領域獲得毎に分割された事になる。

しかし、この方法ではＧＣが総てのデータを新空間にコピーする前に、リスト処理側が動作するのでリスト処理側にも変更が必要になる。つまり、アクセスしたデータが、旧空間中のすでにコピーされたデータかどうか調べ、もしそうならば、forwarding pointerをさらに一段たどらなければ正しいアクセスにならない。

この矛盾が出ないように、Baker の方法ではリスト処理に次のような変更を加えている。

> リスト処理では常に、新空間へのポインタしか見ない。

つまり、リスト処理でデータにアクセスした時に、いつも旧空間へのポインタかどうか調べて（このチェックをデータ値の空間チェックと呼ぶことにする。）、もしそう

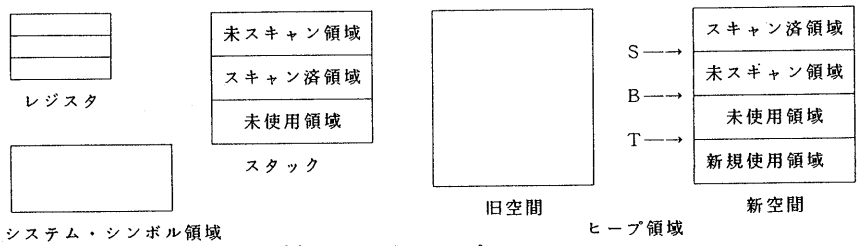

図１　メモリ・マップ

ならば新空間へコピーを行い、その結果をアクセス値とする。こうする事により、アクセスした値が常にＧＣ終了後のものにできる。

また、最新のflip後、新たに獲得された領域には、必ず旧空間中のデータへのポインタは格納されないので、ＧＣのスキャン対象にする必要がない。従って、新たに獲得する領域を、旧空間からのコピー先の領域と別領域（新規使用領域）にすることができる。このことにより、ＧＣによりスキャンされるデータを減らし、ＧＣを早く終了させることができるようになる。

3．Bakerの実時間ＧＣのオーバーヘッド

Baker の方式の一括型コピーＧＣに対するオーバーヘッドは、次の三つに分けられる。

1）データ・アクセス時のオーバーヘッド

データへのアクセス時、アクセスしたデータが、常に新空間中にあるように制御するためのオーバーヘッド。つまり、データ値の空間チェックを行い、必要ならばコピーを行う。

データ値の空間チェックをソフトウェアで実現すると、次のような処理になる。

①タグと値の切り出し

②タグ判定（ポインタかどうかの判定、タグの種類によるマルチウェイ・ジャンプ）

③アドレス判定（旧空間かどうかの判定、領域の上限および下限との比較）

2）コピー制御のためのオーバーヘッド

コピーを K回で止めるためのコピー回数制御のオーバーヘッド

3）スタックの未スキャン領域、スキャン済領域を管理するためのオーバーヘッド

関数からの復帰時にスタックのスキャン境界を示すポインタの管理のオーバーヘッド

この内、最も大きなオーバーヘッドは、データ・アクセス時のオーバーヘッドである。CAR,CDR などの基本関数を含め、処理系の大部分でオーバーヘッドが発生し、非常も大きいものである。しかし、コンパクションを行う実時間ＧＣの場合、アクセス時になんらかのチェックを行うことは　、本質的に必要なことである。

4．改良アルゴリズム

これまでも、Baker の方式のオーバーヘッドを減らす工夫がなされてきている。しかし、そのために空間的なオーバーヘッドが増加したり(2)、仮想記憶環境下に限定されたりしていた(3)。ここでは、アクセス時におけるオーバーヘッド（上記①～③）の軽減を、主な目標にアルゴリズムの改良を行った。

4．1データ・アクセス時のオーバーヘッドの軽減

新空間は、未使用領域、新規使用領域と、現在実行中のＧＣによりスキャンされたか否かによってスキャン済領域と未スキャン領域に分けられる。この領域の境は、ＧＣ中変化し、それぞれレジスタＢ、レジスタＴ、レジスタＳにより指されている。同様に、スタックもスキャン済領域、未スキャン領域、未使用領域に分けられる。

スキャン済領域には旧空間へのポインタは存在しないので、スキャン済領域のデータへのアクセス時、データ値の空間チェックを行うことは、明らかに無駄である。また、アクセス・データが未スキャン領域に存在する場合でも、データの代入先が、やはり未スキャン領域ならば、コピーを行わなくてもよい。（後で、ＧＣがスキャンした時にコピーすることが保証されているから。）これらの場合の判定は、レジスタＳとアクセス・アドレスとの比較のような非常に軽い処理で行えるので、データ値の空間チェックの前にこの判定を入れることによりオーバーヘッドを軽減でき

る可能性がある。つまり、アクセス時、次の判定を初めに行う。

未スキャン領域のデータをスキャン済領域のデータに代入するか調べる。

このチェックをデータの存在領域チェックと呼ぶことにする。

データ値の空間チェックは、アクセスするデータの内容を調べるのに対して、データの存在領域チェックは、データのアドレスを調べる。（図２参照）

データ・アクセス時にデータの存在領域チェックをまず行うと、データ値の空間チェックの必要性は、表１のようになる。

改良方式では、表１に従ってデータの存在領域チェック後、データの値の空間チェック、さらにデータのコピーを行うかを決定している。

未使用領域が充分にあって、実際にはデータのコピーが行われていない時、すべての領域をスキャン済領域と考えることにより、この期間、データ値の空間チェックは一切行わずにデータの存在領域チェックだけに、データ・アクセス時のオーバーヘッドを抑えることが出来る。

また、例えば、関数呼び出しに伴いスタックからレジスタにロードする時のように連続領域からポインタを次々に得る場合には、ポインター一つ一つがどの領域に属するか調べるのでなく、その連続領域が完全にスキャン済領域に属することを調べることにより、すべてのポインタに対してデータ値の空間チェックが不必要であることが判る。このように、データの存在場所でコピーの必要性を判断することにより、一つ一つのポインタでなく、多くのポインタに対して判断を一度に行なえる場合が生じ、オーバーヘッドの軽減を達成することができる（図３参照）。

４．２非ポインタ・オブジェクトの扱い

ストリングや浮動小数のような非ポインタ・オブジェクトは、ＧＣのスキャン対象にする必要がない。従って、これらのオブジェクトのコピー先は、新規使用領域にすることができる。このことにより、ＧＣを早く終了させることができるのみならず、ＧＣのスキャン対象領域にあるオブジェクトは、すべてポインタ幅の要素になるので、ＧＣのスキャンは、ポインタの幅ごとに進めればよくなり、ＧＣルーチンも簡素化できる。

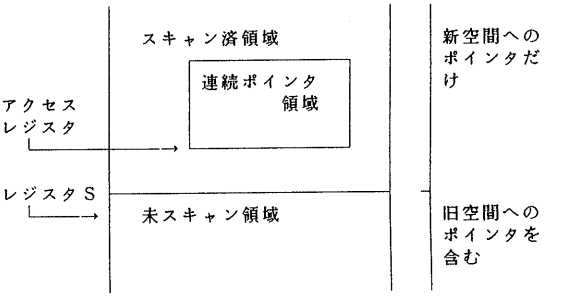

図３　連続領域に対する判定

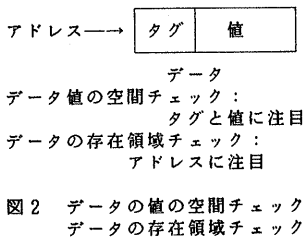

図２　データの値の空間チェックと
データの存在領域チェック

表１　データの存在領域チェックに対する
データの値の空間チェックの必要性

代入先 / 代入元	スキャン済領域	未スキャン領域
スキャン済領域	不必要	不必要
未スキャン領域	必要	不必要

4．3可変長オブジェクトの扱い

ベクタやストリングのような可変長のオブジェクトを、オブジェクト単位でコピーすることは、予想できないリスト処理の中断を招く。これは、実時間性が確保できないことを意味する。そこで、今回のインプリメントでは、ある値VK以上の長さのオブジェクトは、分割してコピーしている（図4参照）。

可変長オブジェクトは、コピー開始時に長さを調べ、VK以上であれば、初めのVK個の要素だけをコピーする。コピー元の第一要素と第二要素には、それぞれforwarding pointer とコピーした要素の数を代入する。コピー先の最終要素には、backward pointer としてコピー元のアドレスを代入する。分割コピー中のオブジェクトは、そのコピー元、コピー先とも、タグ中に”コピー中”のフラグを立てる。再びコピーを始める時には、このビットでコピー中であることが判り、コピーもとの第二要素より、次にコピーすべき要素が判る。総ての要素がコピーされたならば、”コピー中”のフラグを下げる。

ベクタやストリングの要素へのアクセスは、コピー中のビットを調べ、分割コピー中ならば、コピー元の第二要素（コピー先からコピー元へは、コピー先の最終要素により手繰ることができる。）の値と比べることにより、求める要素がコピー先にあるのかコピー元にあるのかを知り、正しい方にアクセスすることになる。

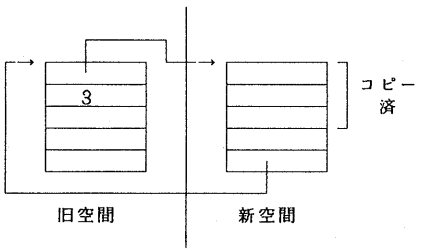

図4可変長オブジェクトのコピー

5．性能

一括型コピーＧＣ（Traditional GC）、Baker の方式（Baker ）および改良方式（Improved）のそれぞれに対してＧＣの性能を測定した。ただし、Baker の方式においては、4．2，4．3でのべた非ポインタ・オブジェクトのの扱いと可変長オブジェクトの扱いを取り入れている。

なお、プログラムは、リスプ・コンテストから選び、計算機はSUN1、システム記述言語は、Ｃ言語である。Lispの仕様としては、Uti Lispに準拠している。

ここでは、ＧＣの性能評価の指標として次のものを測定した。

①空間的効率として最低必要なヒープ領域量

②実時間性として最大中断時間

③時間的効率として全実行時間

以下、それぞれの項目にたいして、一括型コピーＧＣに対する実時間化の効果とそのためのオーバーヘッドに注目して結果を述べる。

①最低必要なヒープ領域量

flipが起って、総てのデータを新空間にコピーし終った時に使用していたヒープ領域量である。実際には、コピー法なので、別空間に対応する領域も必要である。結果を図5に示す。Baker 方式と改良方式は、同じ特性を示すのでReal Time GCとして示す。必要ヒープ量は、プログラム実行中ほとんど変化しない。実時間化することにより、K に反比例して領域が、必要になる。つまり、K=10では、一括型ＧＣに比べて、約 1.2倍程度で済むが、K= 2では、約 2倍になっている。これは、別空間を考慮すると、実効的なヒープの大きさは、実際のヒープの1/4 程度になっていることになる。

②最大中断時間

実時間ＧＣによる最大中断時間を、図６に示す。Bakerの方式と改良方式は、同じ特性を持つ。最大中断時間は、完全に KおよびVKにより制御できる。

一方、一括型ＧＣでは、図６に示すようにアクティブ・データの量に比例して中断時間が延びてしまっている。

③時間的効率として全実行時間

領域の大きさを、K=4, VK=50 の時に実時間ＧＣが必要とする最低量の3/2 倍の大きさとして測定した。一括形ＧＣとの実行時間比を図７に示す。Baker の方式では、約2.6倍の実行時間であるが、改良方式では約1.5 倍程度に抑えられている。また、Baker の方式では、実行時間は、ほとんど Kの値によらない。これは、Baker の方式のオーバーヘッドの大部分が、データアクセス時のものであることを示している。改良方式では、K の値が大きくなるにつれて実行効率が良くなっている。これは、K が大きくなり両空間を使用して動く割合が小さくなったことにより、データアクセス時にデータの存在領域チェックだけで済む割合が増えたことによる。

領域をできるだけ小さくとって、いつもコピーが起きている状態と領域を大きくとりflipが起きない状態での実行時間を、表２、表３に示す。

いつもコピーが起きている状態の実行では、データの存在領域チェックとデータ値のチェックの両方のチェックを一回のデータアクセス時に行う場合が出る。この場合は、データの存在領域チェックがオーバーヘッドとなってしまう。

実際のプログラム実行において、コピーをしている場合二つのチェックを行ってしまう割合を調べると表４のようになる。データのコピー中でも、ほとんどの場合は、データの存在領域チェックのみで、以後の実行を続けることができているのが判る。

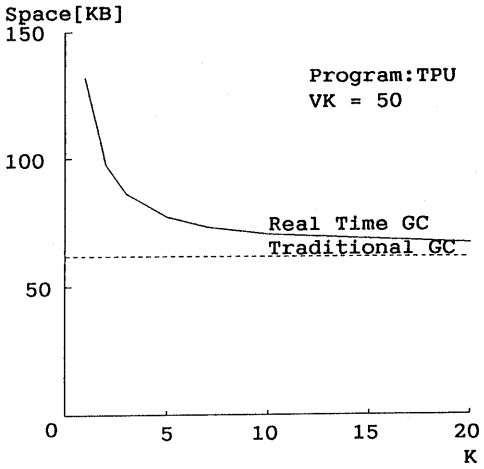

図４　必要ヒープ量

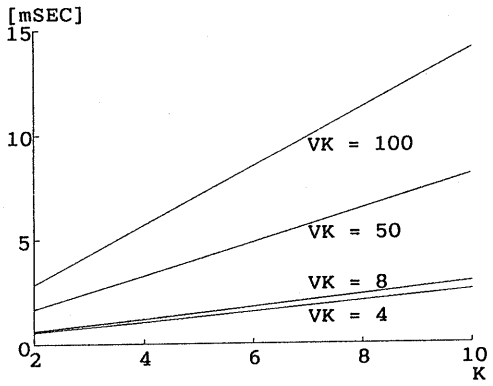

図５　最大中断時間（実時間ＧＣ）

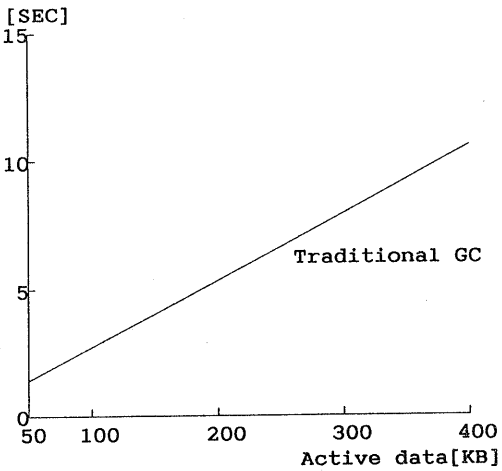

図６　最大中断時間（一括型ＧＣ）

5．まとめ

1）改良方式とBaker 方式との実時間化オーバーヘッドの比

データのコピーが実際には行われていない場合、改良方式においては、Baker 方式に比してそのオーバーヘッドを約30% に減らすことができた。

データのコピーをしている場合には、データの存在領域チェックだけで済まずにデータの値の空間チェックも行う場合が発生する。つまり、データの存在領域チェックがオーバーヘッドとなる場合が起きる。故に、データコピー中には、Baker 方式に対するオーバーヘッドの減少は、 50% 程度に留まっている。ただし、コピー中においても、二つのチェックを行っしまうオーバーヘッドは、データの存在領域のチェックで得られた効果を越えていない。

ソフト・ウェアのみで実時間化を実現する場合、この方式により、アクセス時のオーバーヘッドをBaker 方式の約1/2 以下に軽減することが出来る。特に、アクティブ・データ量に比べ領域が広く、実際には、データのコピーがそう頻繁に行われていない場合に、その効果が高い。

2）改良方式と一括型ＧＣとの全実行時間の比

一括型ＧＣに比べ改良方式は、領域が充分ある場合で、１．５倍、領域が狭くＧＣが頻発する環境で１．７倍の時間がかかる。

3）実時間性

ワークステーション程度の計算機において、ＧＣによる中断時間を、ミリ単位に抑えることができた。

日頃御指導頂く棚橋部長、林部長代理、並びに研究室諸兄に感謝します。

参考文献

(1) Henry G. Baker: "List Processing in Real Time on a Serial Computer" ,Comm.ACM,21(4),1978

(2) Rodney A. Brooks: "Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware",Conference Record of the 1984 ACM Symposium on LISP and Functional Programming

(3) Jeffrey L. Dawson: "Improved Effectiveness from a Real Time LISP Garbage Collection",Conference Record of the 1982 ACM Symposium on LISP and Functional Programming

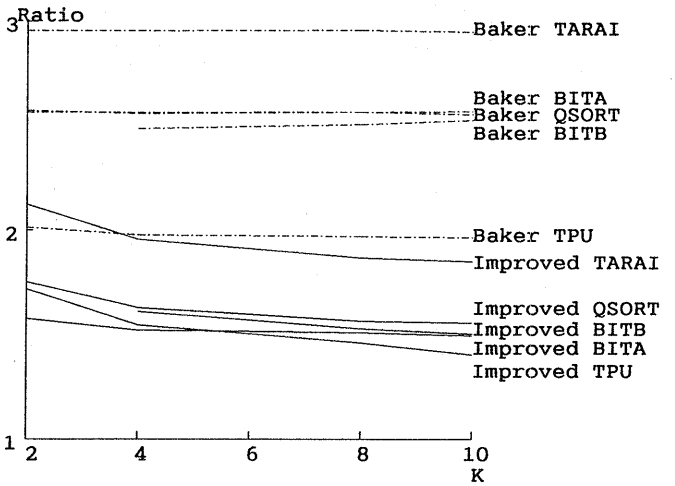

図７　実行時間比

表 2　全実行時間(データ・コピー中　K=4, VK=50)

	BITA	BITB	QSORT	TARAI	TPU
一括型　Tm2 =Tm + Tb2 -Tb1	46.0 (1.0)	12.7 (1.0)	124.6 (1.0)	152.6 (1.0)	158.9 (1.0)
Baker 方式 Tb2 (Tb2 / Tm2)	127.4 (2.7)	35.0 (2.8)	318.8 (2.6)	391.7 (2.6)	255.8 (1.6)
改良方式　Ti2 (Ti2 / Tm2)	77.2 (1.7)	21.1 (1.7)	214.1 (1.7)	270.9 (1.8)	229.9 (1.4)

単位　秒（カッコ内は、一括型に対する比）
BITA　(bita '(a b c d e f g h))　を 5回反復
BITB　(bitb '(a b c d e f g h))　を 5回反復
QSORT (qsort 100要素のリスト)　を 5回反復
TARAI (tarai 8.0 4.0 0.0)　を 5回反復
TPU　(tpu1)　を 実行

表 3　全実行時間(データ・コピーなし)

	BITA	BITB	QSORT	TARAI	TPU
一括型　Tm	44.6 (1.0)	12.0 (1.0)	114.2 (1.0)	146.6 (1.0)	57.0 (1.0)
Baker 方式 Tb1 (Tb1 / Tm)	126.0 (2.8)	34.3 (2.9)	308.4 (2.7)	385.7 (2.6)	156.9 (2.8)
改良方式　Ti1 (Ti1 / Tm)	64.6 (1.5)	17.5 (1.5)	163.3 (1.4)	209.2 (1.4)	87.0 (1.5)

単位　秒（カッコ内は、一括型に対する比 ）
BITA　(bita '(a b c d e f g h))　を 5回反復
BITB　(bitb '(a b c d e f g h))　を 5回反復
QSORT (qsort 100要素のリスト)　を 5回反復
TARAI (tarai 8.0 4.0 0.0)　を 5回反復
TPU　(tpu1)　を 実行

表 4　改良方式で二つのチェックを行う割合（コピー中）

	BITA	BITB	QSORT	TARAI	TPU
二つのチェック を行う割合	10 %	12 %	20 %	20 %	38 %