実時間GCの実現方式と評価†

小沢年弘†† 林 耕司†† 服部 彰††

人工知能用言語に代表される，領域管理をガーベジ・コレクション(GC)で行うシステムで，実時間処理を行わせる場合，GCによる処理の長時間の中断をさける必要がある．従来から知られているコピー法に基づく実時間GC(Bakerの方式)は，実時間化に伴うデータアクセス時のオーバヘッドが大きく，全体の実行時間が著しく長くなる欠点があった．本論文では，実行効率をそれほど落とさないように改良したコピー法に基づく実時間GCの方式を提案し，それを試作し，評価した結果を述べる．本方式では，データアクセス時における実時間化のオーバ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にそのポインタの存在するアドレスを利用することにより達成されている．この方式をLisp言語処理系上にインプリメントし，従来のBakerの方式および一括型コピー法GCとの性能比較を行った．その結果，Bakerの方式に比べ実時間化のオーバヘッドは，約50%以下になっており，一括型コピー法を採用したLisp言語処理系の実行時間と比べても，約1.5倍程度に抑えられている．

1. はじめに

人工知能用言語に代表される，領域管理をガーベジ・コレクション(GC)で行うシステムで，プロセス・コントロール等の実時間処理を行わせる場合，システムの処理が長時間中断することが問題となる．この問題が，これらのシステムが，FAなどの応用に使用されない原因であり，その適用範囲を狭めさせている．また，この長時間の中断が，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[1),2)]．しかし，GC専用ハードウェアのない汎用機においては，実時間化に伴うオーバヘッドが大きく，一括型GCを備えた処理系に比べ効率的な処理系が開発されていない．

本論文では，一括型コピー法GCと従来から知られているコピー法に基づく実時間GC(Bakerの方式)の性能を，汎用機上におけるLisp言語処理系において比較する．さらに，汎用機上でも十分に効率的な実時間化を目指して，実時間化のオーバヘッドを減らしたコピー法に基づく実時間GCの改良方式を提案し，その性能評価を述べる．

† An Implementation and Evaluation of Real Time GC by TOSHIHIRO OZAWA, KOHJI HAYASHI and AKIRA HATTORI (Fujitsu Laboratories Ltd.).

†† (株)富士通研究所

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，一括型GCであるコピー法GCの修正としてBakerにより提案された[3)]．

一括型コピー法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たしGCが起動されると，新空間と旧空間を入れ換え(この処理をフリップと呼ぶ．)，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**)．

(1) レジスタやシステム変数が指すデータを新空間の先頭にコピーする．これらのコピーの始まりとなるものをルートと呼ぶ．

(2) レジスタSに新空間の先頭を，レジスタBに新空間の空き領域の先頭をポイントさせる．

(3) 新空間をスキャンするように，レジスタSを更新して行き，次にコピーされるべきデータを探していく．コピー先は，レジスタBがポイントしている領域であり，レジスタBは常に新空間の空き領域をポイントするように更新される．

(4) 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．

コピーする際，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，ホワーディング・ポインタと呼ぶ．)．そして，一度コピーしたデータへのポインタは，ホワーディン

グ・ポインタのポイント先に付け変えられる.

Baker の方式でも新空間の未使用領域を使い尽くすと，フリップを起こし，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．残りのデータのコピーは，領域獲得ごとに，やはりK個ずつ行われる．つまり，GC は，領域獲得ごとに，分割されたことになる.

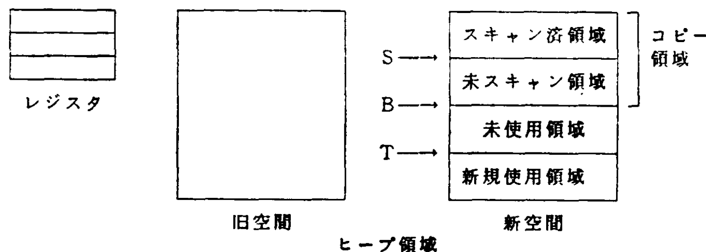


図 1 メモリ・マップ
Fig. 1 Memory map of copying GC.

しかし，この方法では GC がすべ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，リスト処理側が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，ホワーディング・ポインタ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．このような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ようにする．つまり，リスト処理でデータにアクセスした時に，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．こうすることにより，アクセスした値は常に新空間のものとなり，GC 終了後のものにできる．また，フリップ後，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．したがって，新たに獲得するデータは，旧空間からのコピー先の領域と別領域（この領域を新規使用領域と呼ぶ.）からにすると，GC によりスキャンされるデータ量を減らすことができる．これにより，GC を早く終了させることができる.

3. Baker の方式のオーバヘッド

Baker の方式の一括型コピー GC に対するオーバヘッドは，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・タグ方式において，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ヘッド

コピーを K 個で止めるためコピー回数制御のオーバヘッド

（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ヘッド

このうち，最も大きなオーバヘッドは，データ・アクセス時のオーバヘッドである．これは，car, cdr などの基本関数を含め，処理系の大部分で発生するので，非常に大きいものである．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. 改良方式のアルゴリズム

これまでも，Baker の方式のオーバ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ヘッドが増加したり[4]，仮想記憶環境下に限定されたりしていた[5]．ここでは，アクセス時におけるオーバヘッド（上記①〜③）の軽減を，主な目標に方式の改良を行った.

4.1 データ・アクセス時のオーバヘッドの軽減

新空間は，コピー領域と未使用領域と新規使用領域とに大きく分けられる．さらに，コピー領域は，現在実行中の GC によりスキャンされたか否かにより，スキャン済領域と未スキャン領域に分けられる．これらの領域の境は，GC 中変化，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．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．これは，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などの実時間化のための処理を行う必要がないことを意味している．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，そのデータを，やはり未スキャン領域のデータに代入するのならば，アクセス時に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である．このように，アクセスするデータの存在する領域により，実時間化のための処理を行わなくてよい場合が生じる．

データがどの領域に存在するかの判定は，レジスタSとアクセス・データ・アドレスとを比較すればよいので，データ値の空間チェックよりも非常に簡単な処理で行える．したがって，データがどの領域に存在するかをチェックし，必要ならば，データ値の空間チェックなどの処理を行うようにすれば，オーバヘッドを軽減することができる可能性がある．つまり，データ・アクセス時，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の判定を始めに行う．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・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．

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）．

改良方式では，データの存在領域チェック後，表1に従って，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．

この方式では，未使用領域が十分にあって，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，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ヘッドを抑えることができる．

また，例えば，関数呼び出しに伴いスタッフからセイブした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一度にわかる．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一度に判断を行える場合が生じるので，オーバヘッドが軽減される（図3参照）．

表1 改良方式におけるデータ値の空間チェックの必要性
Table 1 Necessity of checking data value in improved method.

代入元＼代入先	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

4.2 非ポインタ・オブジェクトの扱い

文字列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．したがって，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．なぜならば，新規使用領域の意味は，GCによりスキャンする必要がないオブジェクトの領域ということであるから．これにより，

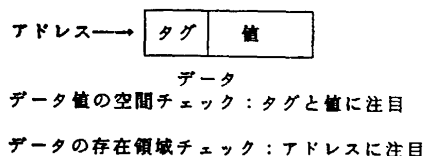


図2 データの値の空間チェックとデータの存在領域チェック
Fig. 2 Data value is checked in Baker's method. Data address is checked in improved method.

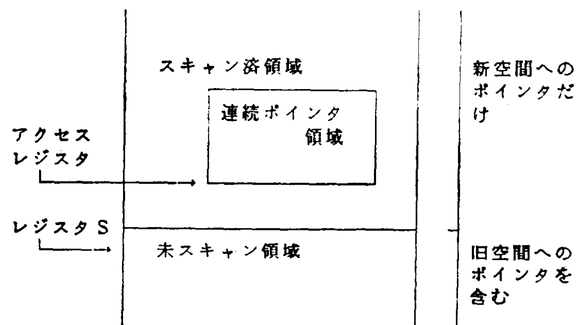


図3 連続領域に対する判定
Fig. 3 Checking of continuous pointer area.

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．

4.3 可変長オブジェクトの扱い

ベクタや文字列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，Baker の論文に示されているように，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．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，領域は確保するが，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，それぞれホワーディング・ポインタ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，バックワード・ポインタ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わかり，コピー元の第二要素より，次にコピーすべき要素がわかる．すべ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．

ベクタや文字列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．

5. 性能評価

次の 3 種類の GC 方式に対して，Lisp 言語処理系で性能を比較した．

（1） 一括型コピー法 GC (stop-and-collect GC)
（2） Baker の方式 (Baker)
（3） 改良方式 (Improved)

ただし，Baker の方式においては，4.2, 4.3 で述べた非ポインタ・オブジェクトの扱いと可変長オブジェクトの

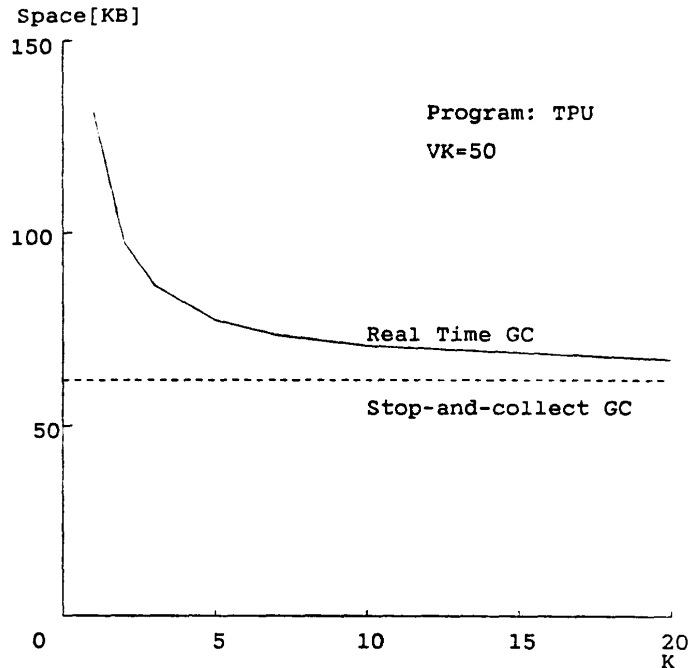


図 4 必要ヒープ量
Fig. 4 Needed heap area size.

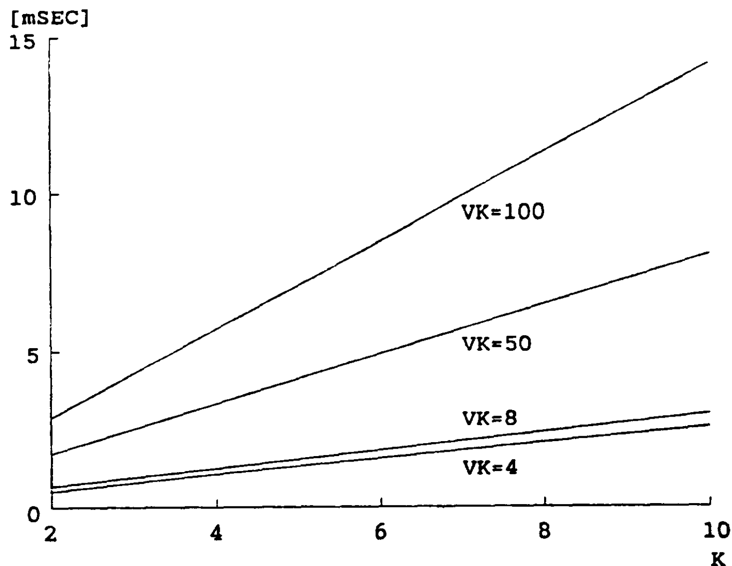


図 5 最大中断時間（実時間 GC）
Fig. 5 Maximum interrupt time by real time GC.

クトの扱いを取り入れている．

プログラムは，リスプ・コンテストから選び，計算機は SUN 1，Lisp 言語処理系は，Uti Lisp に準拠したインタプリタを用いた．言語処理系の記述言語は，C言語である．

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，それぞれの項目に対して，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

フリップが起こって，すべ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．結果を**図 4** に示す．Baker の方式と改良方式は，同じ特性を示すので Real Time GC として示す．実際には，別空間に対応する領域も必要である．必要なヒープ量は，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．

（2） 最大中断時間

実時間 GC による最大中断時間を，**図 5** に示す．Baker 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） 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 倍の大きさという条件で測定した．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 の方式のオーバヘッドの大

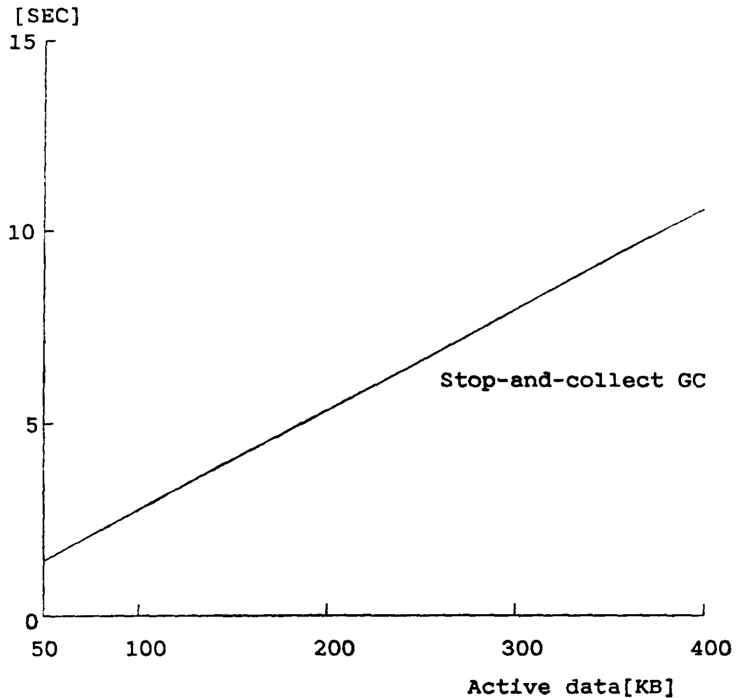


図 6 最大中断時間（一括型 GC）
Fig. 6 Maximum interrupt time by stop-and-collect GC.

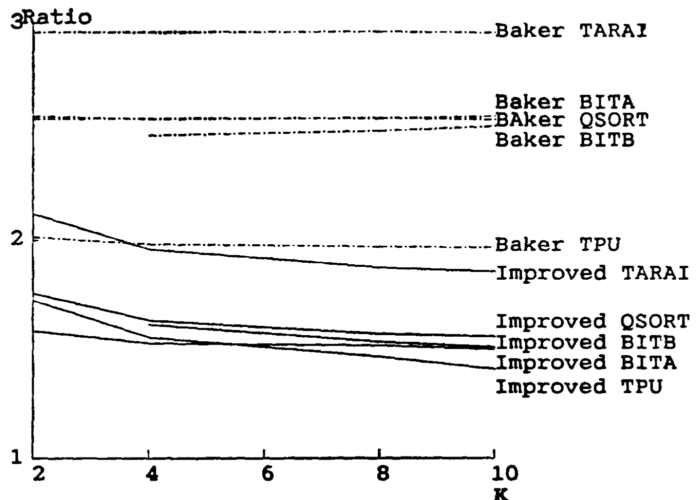


図 7 実行時間比
Fig. 7 Ratio of execution time.

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ることにより，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ることによる．

また，領域をできるだけ小さくとって，いつもコピーをしている状態と領域を大きくとりコピーが起きない状態での実行時間を，**表 2**，**表 3** に示す．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，結果的にオーバ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るのがわかる．

表 2 全実行時間（データ・コピー中 $K=4$, $VK=50$）
Table 2 Execution time in small heap.

	BITA	BITB	QSORT	TARAI	TPU
一 括 型 Tm2=Tm+Tb2−Tb1	46.0 (1.9)	12.7 (1.0)	124.6 (1.0)	152.6 (1.0)	158.0 (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 (tpu 1) を実行

表 3 全実行時間（データ・コピーなし）
Table 3 Execution time in large heap.

	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)	82.0 (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 (tpu 1) 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）
Table 4 Ratio of checking data value in improved method.

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

6. まとめ

1) 改良方式と Baker の方式との実時間化オーバヘッドの比

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ヘッドとなる場合が起きる．したがって，データコピー中には，Baker 方式に対するオーバヘッドの減少は，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てしまうオーバヘッドは，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，改良方式は，実時間化のオーバヘッドを Baker 方式の約 1/2 以下に軽減することができる．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，領域が十分ある場合で，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．

謝辞 日頃御指導いただく棚橋部門長，林部長，ならびに研究室諸兄に感謝します．

参 考 文 献

1) Cohen, J. : Garbage Collection of Linked Data

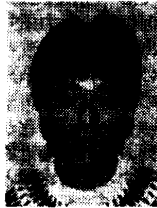
Structures, *ACM Comput. Surv.*, Vol. 13, No. 3, pp. 341-367 (1981).

2) Moon, D. A.: Garbage Collection in a Large Lisp System, *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pp. 235-246 (1984).
3) Baker, H. G.: List Processing in Real Time on a Serial Computer, *Comm. ACM*, Vol. 21, No. 4, pp. 280-294 (1978).
4) Brooks, R.A.: 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*, pp. 256-262 (1984).
5) Dawson, J. L.: Improved Effectiveness from a Real Time LISP Garbage Collection, *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pp. 159-167 (1982).

(昭和62年12月16日受付)
(昭和63年3月9日採録)

小沢 年弘（正会員）

昭和35年生．昭和57年横浜国立大学工学部電気工学科卒業．昭和59年同大学院修士課程修了．同年，(株)富士通研究所入社．並列計算機，並列言語等に興味を持つ．



林 耕司

1960年生．1985年電気通信大学大学院応用電子工学専科修了．同年，(株)富士通研究所入社．言語処理系，並列計算機等に興味をもつ．

服部 彰（正会員）

昭和24年生．昭和47年大阪大学工学部電子工学科卒業．昭和49年同大学大学院工学研究科修士課程修了．同年(株)富士通研究所入社．現在，人工知能研究部第3研究室長．記号処理マシン，並列処理マシンの研究開発に従事．電子情報通信学会，人工知能学会各会員．