ICOT Technical Report: TR-498

TR-498

並列論理型言語KL1の多重参照管理によるガーベジコレクション

木村　康則、近山　隆

August, 1989

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

並列論理型言語KL1の多重参照管理による ガーベジコレクション

木村康則* 近山 隆

(財) 新世代コンピュータ技術開発機構

*1989年4月1日より（株）富士通研究所に帰属.

梗概

本論文では，並列論理型言語 KL1 のデータの参照数管理を行うことによるインクリメンタル GC 方式を提案し，その方式の評価結果を報告する．

ICOT では，並列推論マシン PIM の研究開発を行っている．PIM 上の言語は，並列論理型言語 KL1 である． KL1 は副作用を持たない言語であるため，単純に実装すると，メモリを単調かつ急激に消費してしまい，ガーベジコレクションを頻繁に引き起こす．そこで，本論文では，データ自体ではなく，そのデータへのポインタにデータの参照数を示す 1 ビットのフラグを設け，このフラグをコンパイラと処理系の両方で管理することにより，参照が無くなった時点で積極的にデータを回収するインクリメンタル GC 方式を提案する．構造体データへのポインタのフラグは構造体データの参照数が '1' か '2' 以上かを表し，未定義変数へのポインタのフラグは未定義変数の参照数が '2' 以下か '3' 以上かを表す．本方式は，参照数情報をポインタ側に持つため，参照数更新のための余分なメモリ参照が不要になり，参照数更新の手間が小さくなる．従って，本方式はマルチプロセッサ上での実現が容易である．

汎用計算機上に作成した処理系による評価の結果，インクリメンタル GC を行わない場合に比較して，コード量の増加も僅かで，多くの場合，ヒープ量は半分程度でプログラムを実行できることが分かった．また，回収されるデータの生存期間は非常に短くキャッシュ機構を持つプロセッサでは，ヒット率の向上も望めることが分かった．さらに，一括型 GC の回数も大幅に減らせることが分かり，実行時間についても全体のメモリ量に対してアクティブセルの割合が大きいほど，本方式による GC が有利であることを示した．

1 はじめに

ICOTでは，第五世代コンピュータプロジェクトの一環として並列推論マシンPIMの研究開発を行っている．PIM上の言語KL1[3]は，制限を加えたGHC(Guarded Horn Clauses)[11]すなわち，フラットGHCを基本とした言語で，非同期通信機能を備えた並列論理型言語の一種である．フラットGHCは副作用を持たないため，プログラミングやデバッギングが楽になると期待される半面，pure Lispやと同様に，メモリの消費速度が速く，ゴミ集め(ガーベジコレクション，以下GC)を頻繁に起こす．

GCの実現方式については，pure Lispに始まる関数型言語の処理系に関して多く提案されており，メモリ領域を使い切った時点で再利用可能領域を回収する一括方式，データに参照数のカウンタを設け，それが零になった時点で回収するリファレンスカウント方式(RC方式)などが代表的である[4, 5]．また，一括GCの際の実行の中断を避けるために，インクリメンタルにGCを行う一括GC方式の変形改良型も提案されている[1, 16]．しかし，これらの方式をマルチプロセッサ上で実現しようとすると，単一プロセッサの場合に比べて，動的なオーバヘッドが大きくなる．すなわち，マルチプロセッサ上のGCは，プロセッサ間で共有されたデータ(疎結合システムの場合)や，共有メモリ上のデータ(密結合システム)を扱わなければならない．これは，一括GC方式では実行の中断時間(GC時間)を長くし，RC方式では参照カウンタの一回毎の更新の手間を大きくする．更に，一括型GCはGCの局所性が無く，GC時にはキャッシュのミスヒットやページフォールトが増える．従って，マルチプロセッサ上のGCとしては，実行時オーバヘッドが少なく，かつシステム全体のGCを行う回数を出来るだけ少なくするような方式が望まれる．

そこで，本論文では，データ自体ではなく，そのデータへのポインタにデータの参照数を示す1ビットのフラグを設け，このフラグをコンパイラと処理系の両方で管理することにより，小さな実行時オーバヘッドで多くの場合に良好な回収効率を実現するインクリメンタルGC方式を提案する[2, 19, 13, 17]．構造体データへのポインタのフラグは構造体データの参照数が'1'か'2'以上(多数)かを表し，未定義変数へのポインタのフラグは未定義変数の参照にが'2'以下か'3'以上(多数)かを表す．データへの参照の増減は，コンパイラが調べ，参照が増える場合にはフラグを'多数'にし，参照が無くなり回収できる可能性がある場合には回収用の命令をコンパイルコード中に生成して実行時の処理を軽減する．一方，処理系では，

回収用の命令を実行すると，回収対象データのフラグを調べ，真に回収できるかどうかを決める．なお，本方式では，一旦‘多数’参照となったデータに対しては参照数を減らす操作は行っていないため，後で参照が無くなっても回収出来ない．従って，一括型 GC が別途必要となる．

次章以下では，まずインクリメンタル GC 方式の詳細を述べ，次に単一プロセッサ上に作成した処理系エミュレータを用いて行った実験および評価結果について報告する．

2 並列論理型言語 KL1

2.1 シンタックス

並列論理型言語 KL1 は，フラット GHC に基づいて ICOT で設計された言語である．KL1 プログラムは，次のようなシンタックスを持つ節（またはクローズ）の集合として表される．

$$H :- G_1, \cdots, G_m | B_1, \cdots, B_n. \ (m \geq 0, n \geq 0)$$

ここで，H, G_i, B_i は，各々，クローズヘッド，ガードゴール，ボディゴールと呼ばれる．‘|’ はコミットメントオペレータと呼ばれ，クローズ中でこれに先立つ部分を受動部（またはガード部），これに続く部分を能動部（またはボディ部）と呼ぶ．ここで，ガードゴールには，組込述語しか書けない．これは GHC の言語記述能力を保ちながら，効率的な実現を考慮して採用された制限である．また，ボディゴールには，組込述語とユーザ定義述語の両方のゴールが書ける．

2.2 実行方式

本節では，図 1 に示す様な，整数を要素とするリストから，ある与えられた整数の倍数を取り除くプログラムを使って KL1 プログラムの実行方式について説明する．述語‘filter/3’のクローズヘッドの第一引数は取り除く基になる整数，第二引数は整数のリスト，第三引数は結果のリストが置かれる変数である．プログラムでは，第二引数のリストを‘Car’，‘Cdr’に分解し，‘Car’が第一引数の整数の倍数かどうかをガード部の組込述語で調べる．もし倍数なら，‘Car’を捨て（クローズ (1)），倍数でなければ結果のリストに入れる（クローズ (2)）．クローズ (3) は，整数リストの終端条件である．この例では，ガード部でリストの分解が行われ，

```
filter(P, [X|Xs0], Ys0) :- X mod P =:= 0 |
                           filter(P, Xs0, Ys0).    (1)
filter(P, [X|Xs0], Ys0) :- X mod P =\= 0 |
                           Ys0 = [X|Ys1],
                           filter(P, Xs0, Ys1).    (2)
filter(P, [], Ys0) :- true | Ys0 = [].             (3)
```

図 1: フィルタプログラム例

もしこのリストへの参照数が '1' であれば, コンスセル[1]は分解後捨てられる. 一方, クローズ (2) が選択された時には, ボディ部では新たなリストの生成が起こる. 従ってこの場合には, リダクション毎に 1 個のコンスセルの解放, 割り付けが起こることになる.

2.3 ガーベジコレクション方式

前節 (2.2 節) で述べた様に, KL1 ではメモリの解放, 新たな割り付けが頻繁に起こり, メモリを急速に消費する. 副作用を許す言語では, プログラマの責任で領域を破壊的に使うことにより, Prolog の様にバックトラックのある言語では, バックトラックにより解放された領域を処理系によって暗黙の内に再利用することによって, この急激なメモリ消費を抑えることができる. しかし, KL1 は副作用を許さない言語であるため, プログラマが陽にメモリ管理をすることが不可能である. そのため, 単純に実装すると, メモリ領域を単調かつ, 急速に消費する. たとえば, ある構造体の一部の要素を次々と書き換えながら何回も使おうとすると, 元々の構造体の一部の要素が異なった新しい構造体が次々と生成されてしまう. このようなメモリの急激な消費は, GC を頻発させる. さらに, メモリアクセスの局所性が悪くなることからキャッシュのミスヒットやページフォールトも多くなる. そこで, 実行時に不要になったメモリ領域を実行時に効率的に回収し, 再利用する機構が必要になる. インクリメンタル GC 方式としては, 各データオブジェクトにそこへの参照数を示すカウンタを設ける参照カウンタ方式が一般的である. しかし, この方式では,

[1]リストを構成する 2 語長セル

1. 原理的に一語長分の参照カウンタフィールドがデータオブジェクト毎に必要である.

2. 参照カウンタの更新のためにメモリアクセスが必要でオーバヘッドが大きい.

3. 循環構造データを回収出来ない.

という問題がある. 1. は, 元々一語長のデータについては一時に使えるメモリ領域がコピー方式の一括型 GC と同じく全体の半分程度になってしまうことを意味している. また, KL1 は, 疎結合あるいは密結合マルチプロセッサ上に実現される. この時には, 疎結合の場合には, プロセッサ間に渡ったポインタによってデータが共有され, 密結合の場合には, 共有メモリによってデータが共有される. そして, これらのデータの参照数管理をしようとすると, プロセッサ間で参照数更新のためのメッセージを流したり, 共有メモリの排他制御が必要となる. 従って, 前記 2. の問題点は, マルチプロセッサ上での実現を考えた時には, より顕著になる. 一方, 実際の KL1 プログラムの実行の様子を検討すると, データの被参照数は, '1' の場合が非常に多いことが予想される. また, KL1 におけるユニフィケーションなどでは, データオブジェクト本体をアクセスする必要が無く, オブジェクトへのポインタの操作で済むことが多い[2]. そこで, 次章以降では, データに対するポインタ側にそのデータの被参照数が '1' かそれ以上 (' 多数 ') かを示す 1 ビットのフラグ (Multiple Reference Bit, 以下 MRB) を設け, これによってデータの参照数を管理して GC を行う方式について提案し, 評価を行う.

3 MRB 方式

3.1 MRB 方式の考えかた

MRB は, オブジェクトではなく, ポインタに付随する 1 ビットのフラグである. MRB の値は, 基本的には, ポインタの指す先のデータセルの参照数が '1' であるか, ' 多数 ' であるかを示す. この結果, MRB 方式は, 一般の参照カウンタによる管理に比べて以下のような特徴を持っている.

1. 参照数を管理する情報 (1 ビットのフラグ) が簡単である.

2. 参照数の更新操作が, データオブジェクトにアクセスすることなく可能である.

[2]例えば, 変数と構造体のボディ部でのユニフィケーションでは, 構造体本体へのポインタを変数セルに代入すればよく, 構造体本体へアクセスする必要はない.

これは，2.3 節で述べた一般の参照カウンタ方式の問題点を解決し，マルチプロセッサ上での効率的な実現を容易にするものである．MRB 方式では，あるデータオブジェクトが参照数 '1' で生成され，その後参照数に変化がなければ，そのデータを消費[3]したゴールによってデータオブジェクトを回収できる．参照数の増減は，コンパイラが KL1 プログラムの個々のクローズを解析することによって検出し，参照数が増加する場合には MRB を '多数' にする命令を，データが回収できる可能性のある場合には，回収のための命令を生成する．また，構造体のようなデータオブジェクトの一部を書き換えた構造体を生成する場合，その構造体への参照数が '1' であり，元の構造体はその後は不要であれば，元の構造体の一部を書き換えてそのまま利用できる．ただし，参照数が一旦 '多数' になると，その後は参照するものが無くなっても検出できないため，不要データの回収は不完全になる．また，本方式でも一般の参照管理方式 GC と同じく循環構造を回収できない．そこで，他方式の GC と併用する必要がある．

3.2 MRB の定義

KL1 プログラムの実行において扱うデータオブジェクトは，構造体[4]，未定義変数，定数および間接ポインタに大別できる．これらのデータオブジェクトは，KL1 のゴールまたは構造体の要素として，ポインタ (または間接ポインタの列) を通して参照される．MRB は，ポインタに付加された 1 ビットの参照カウンタに相当する．ただし，定数の場合は，定数自体も MRB を持ち，未定義変数は MRB を持たない．MRB ビットを ○('白い' と呼ぶ) または ●('黒い' と呼ぶ) で表現することにすると，その基本的な意味は以下の様になる．

○：このポインタの指すデータへの参照はこれ一本である (単一参照)．

●：このポインタの指すデータへの参照が他にもあるかも知れない (多重参照)．

ただし，未定義変数セルへのポインタが持つ MRB は扱いが少し異なる．KL1 の未定義変数には，基本的に，それを具体化するための参照と，具体化された値を読み出す参照の二つのポインタがあることが多い[5]．そこで，未定義変数セルへのポインタが持つ MRB の値の意

[3]データの内容を読んで，データオブジェクト自体はその後使わなくなること．

[4]リスト，ベクタおよびストリング

[5]変数は，書き手と読み手があってはじめて意味があるのが普通である．

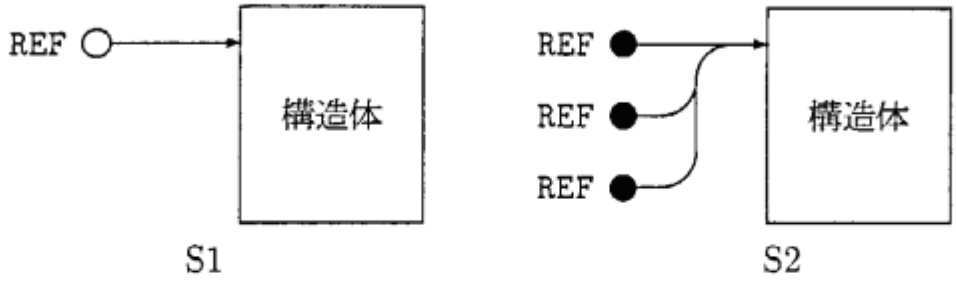


図 2: 構造体の管理

味を以下の様に決める.

○：参照される未定義セルに対するポインタは, 他には白いものが 1 本あるか, 黒いものが任意本あるかのいずれかである.

●：参照される未定義セルに対するポインタが他に幾つでもあるかも知れない.

さらに, 単一化によって具体化した変数セルの MRB の値を以下の様に決める.

変数同士, あるいは変数と構造体の単一化では, 変数あるいは構造体を指すポインタの MRB のうち, どちらか一方が黒であれば, 具体化した変数セルの MRB の値は黒くする. 両方とも白の場合のみ具体化した変数セルの MRB の値も白くなる.

このように MRB の値を意味づけすると, 構造体, 変数セル, および単一化によって具体化した変数セルに対するポインタの MRB の許される組み合わせは, 図 2, 図 3, 図 4のようになる.

3.3 MRB の更新操作

KL1 の実行は以下の様な操作をメモリ領域に対して行いながら進む. このとき, 3.2 節で述べた条件が満たされるように, 各ポインタまたは定数の MRB の管理を行わなければならない.

1. 変数, 構造体などの生成
2. 変数, 構造体などのゴール間にわたる分配
3. 変数のデリファレンス
4. ユニフィケーション

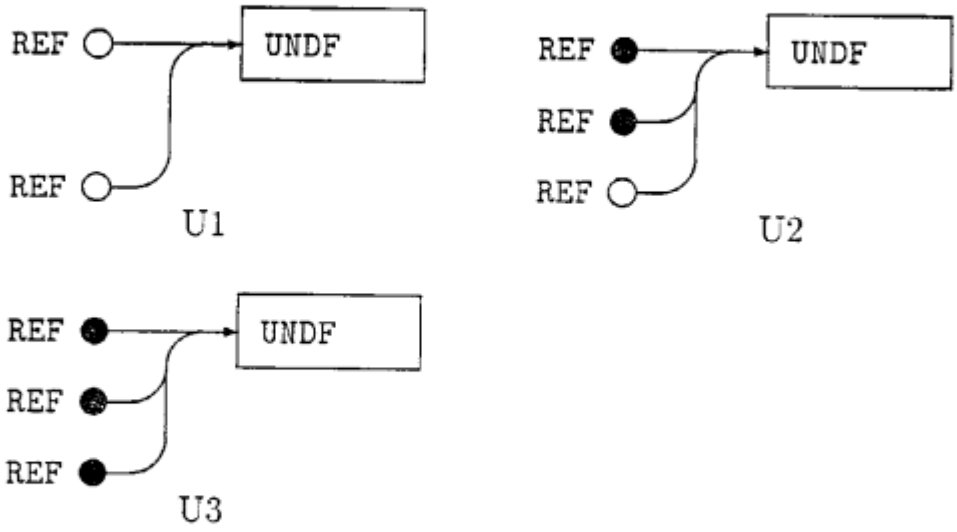


図 3: 未定義変数の管理

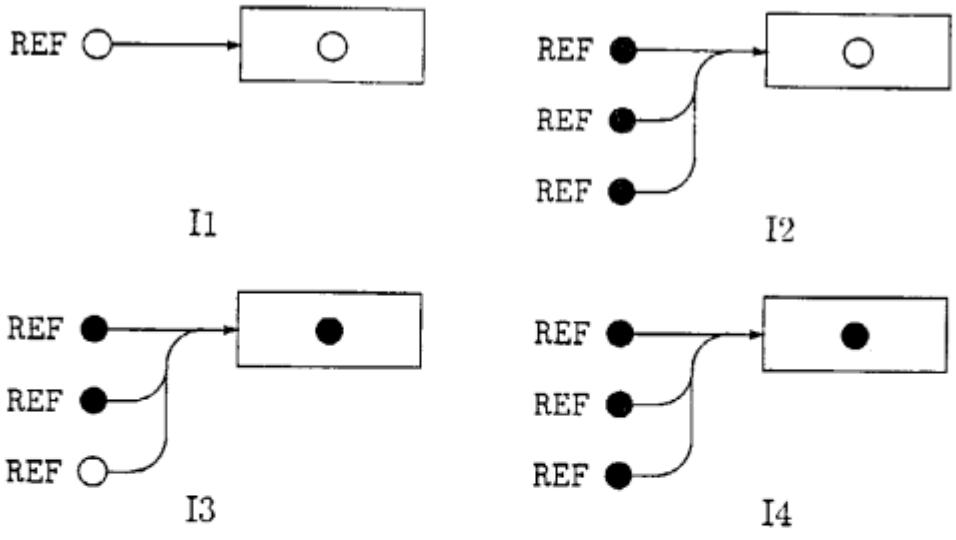


図 4: 具体化変数の管理

5. 構造体要素の取り出し

このうち, 1, 2 はコンパイル時にクローズを解析し MRB 管理命令を生成することによって, 3,4 は実行時に MRB の値を調べることによって, MRB 管理が行われる. 以下に, 個々の場合についての MRB 管理の方法を説明する.

3.3.1 変数, 構造体などの生成

クローズのボディ部で新たに変数セルを割り付けたり, 構造体を作る場合である.

```
p :- true | q(X), r(X, [car|cdr]).        (1)
p :- true | q(X), r(X, [car|cdr]), s(X). (2)
```

クローズ (1) の場合は, ボディ部で新たに割り付ける変数 X の出現回数が 2 なので, このセルに対するポインタの MRB は白である (上記 U1 に相当). 一方, クローズ (2) では, 3 回 (以上) 現れているので MRB は黒となる (U3 に相当). リスト [car|cdr] に対するポインタの MRB はクローズ (1), (2) とも白である。

3.3.2 変数, 構造体などのゴール間にわたる分配

クローズのヘッドで受けた引数をボディ部で分配する場合である.

```
p(X) :- true | q(X).          (1)
p(X) :- true | q(X), r(X).    (2)
```

クローズ (1) では, ヘッドで受けた引数をそのままボディゴールに転送するだけなので参照数の増減はなく, MRB に変化はない. クローズ (2) では, 2 つに分配するので, MRB は黒にする (X が未定義変数であれば U2 または U3 に相当する).

3.3.3 変数のデリファレンス

変数のデリファレンスを行う場合である. デリファレンス結果の MRB はデリファレンス途中の間接ポインタのなかで 1 つでも MRB が黒のものがあれば黒, それ以外の場合は白をデリファレンス結果の MRB とする. この操作は, デリファレンスチェインに現れる間接ポインタの MRB の論理和をとることによって実現される.

表 1: ユニフィケーション時の MRB 管理

$X \setminus Y$	S1	S2	U1	U2 白	U2 黒	U3
S1	⋆	⋆	I1	I1	I3	I4
S2	⋆	⋆	I3	I4	I3	I4
U1	I1	I3	I1	I2	I3	I2
U2 白	I1	I4	I2	I2	I4	I4
U2 黒	I3	I3	I3	I4	I3	I3
U3	I4	I4	I2	I4	I3	I4

3.3.4 ユニフィケーション

ユニフィケーションの場合の MRB の管理は，まず，2つの変数をデリファレンスし，どちらかの MRB が黒であれば黒で具体化，さもなくば白で具体化する．表 1 に個々の場合を示す．変数 X, Y のデリファレンス結果が表の第一行と第一列で表された場合の具体化されたセルの MRB を，対応する行，列位置で示す．表で，⋆ は，構造体同士のユニフィケーションが行われることを示し，この場合には，要素ごとのユニフィケーションがこの表のルールに従って行われる．また，U2 白及び，U2 黒はそれぞれ白のポインタ／黒のポインタを使ってユニフィケーションが行われることを示す．

3.3.5 構造体要素の取り出し

ガード部でベクタの要素を取り出し，その要素とベクタ本体をボディ部に転送する場合である．

```
p(V, I) :- vector_element(V, I, E) | q(V, E).
```

ガード部の組込述語 'vector_element' は，ベクタ V の I 番目の要素を取り出し，変数 E で受け，両方をボディ部へ転送する．この場合，読み出したベクタの要素は，ベクタの本体と，変数 E から指されているので，ベクタ V の I 番目の要素の MRB と変数 E の MRB の両者を黒くしなければならない．

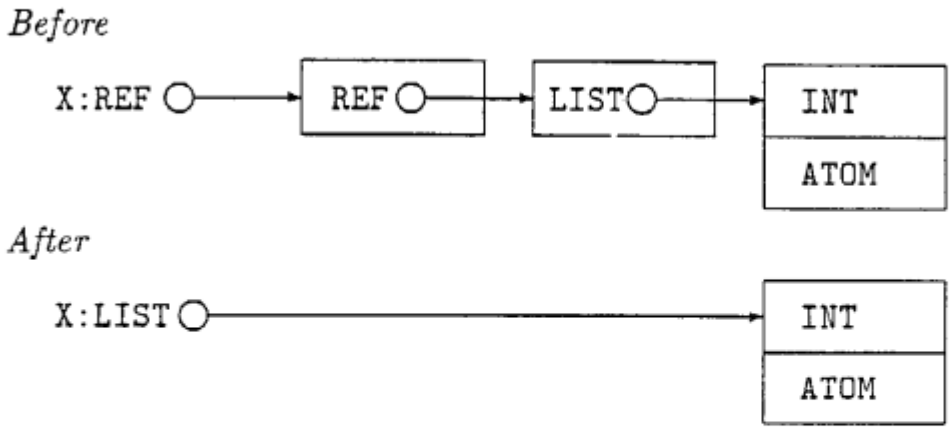


図 5: 間接ポインタの回収

4 MRB によるガーベジコレクション

3.3 節の様に KL1 のデータを MRB 管理すると，KL1 の実行中に，あるデータへの参照が最後でかつ MRB が白であれば，そのデータは回収し，再利用できることになる. 回収が可能な場合について以下に説明する.

4.1 変数のデリファレンス

変数のデリファレンス時の間接ポインタは，その間接ポインタセルへの参照が単一で (ポインタの MRB は白)，この間接ポインタの MRB が白であれば，デリファレンス時に回収できる. これは，間接ポインタの MRB が図 4 の 'I1' の状態にあることを意味し，この場合には，間接ポインタへの参照は '1' であることが保証されるので，回収しても良いことになる. 図 5 で，デリファレンス後は，変数 X から指される間接セルと，コンスセルを指す間接セルは，回収される.

4.2 構造体とのユニフィケーション

受動部で，ゴールの引数の一つが構造体との単一化に成功したとき，ゴール引数として与えられた構造体は MRB が白であれば回収できる. 例えば以下の様なクローズがあったとき，第一引数として与えられたコンスセルは MRB が白であれば回収できる. ベクタが現れた場合も同様である.

```
p([X|Y]) :- true | b(X), c(Y).
```

4.3 ガード部のみに現れる変数とのユニフィケーション

ゴール引数がガード部のみに現れる変数とのユニフィケーションに成功したとき，そのゴール引数の MRB が白であれば回収できる．以下の例で変数 Void はボディ部に現れないので，このクローズが選択されれば変数 Void から指されているデータが回収の対象となる．

```
p(Void) :- true | true.
```

4.4 ボディ部での構造体要素の更新

ユーザプログラム中で使われているテーブル (表) の類いを KL1 のベクタで表し，この要素をボディ部で更新するとき，そのベクタの MRB が白ならば更新する要素をベクタ本体に破壊的に書き込み，ベクタ本体を再利用することが出来る．

```
p(Vold, I, C, Vnew) :- true |
        set_vector_element(Vold, I, C, Vnew).
```

この例では，ボディ部の組込述語 set_vector_element で，ベクタ Vold の I 番目の要素を C に書き換えて新しいベクタ Vnew にしている．もし，Vold の MRB が白ならば，Vold を再利用し，I 番目の要素を C にして Vnew とすることができる．Vold の MRB が黒ならば，Vnew のために新たにベクタを割り付け，I 番目以外の要素を Vold からコピーし，I 番目の要素を C にしなければならない[14]．

4.5 回収の方法とタイミング

4.2 節や，4.3 節で示したようなデータの回収の可能性は，KL1 のソースクローズを解析することによってわかるので，コンパイラによって回収のための命令を陽に出すものとする．また，実際の回収の可否はそのクローズが選択されるとわかった時点で行なわなければならない．従って，回収のための命令は概念的には，コミットメントオペレータの位置に生成される．一方，デリファレンス時の回収は動的にしか分からないので実行時に行われる．

5　実験処理系と命令セット

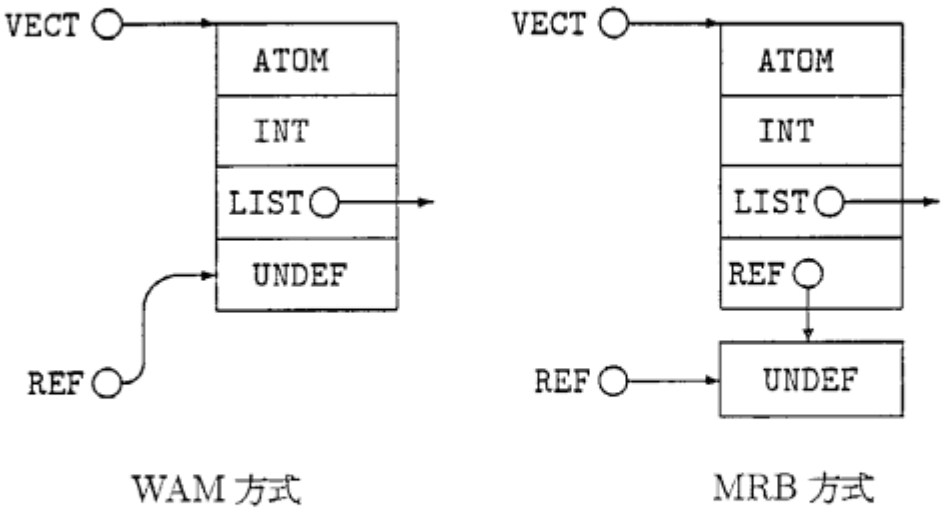
5.1　実験処理系

MRB-GC 方式の有効性と問題点を調べるために, PDSS システム[6] を使って実験を行った. PDSS システムは, UNIX マシン上に構築された単一プロセッサ上での KL1 の実行を行う処理系である. この処理系は, 既に提案した KL1 の抽象マシン[7] に準拠している. すなわち, ユニフィケーションを行うためのレジスタ群 (引数レジスタと呼ばれる) や, 実行環境を保持するための制御レジスタ群を持ち[6], メモリには, MRB のフィールドを持った変数やコンスセルに使われるメモリがそのサイズ毎にフリーリストとして管理されている. これは, MRB 方式では, データの割り付け, 開放が動的に起こるため連続領域を端から使っていくと言ったメモリ管理が適さないことから採られた方法である. また KL1 のゴールは, メモリ上でゴールレコードとして表され, ゴールレコードには実引数を格納するための引数スロット, このゴールを実行するためのコード, その他の制御情報を格納するためのスロットがある. さらに, ゴールの実行が中断した時の処理を行うために使うサスペンションレコードなどがメモリに採られている. KL1 のプログラムは, まず, KL1-B[7] 命令列にコンパイルされる. そして, PDSS 処理系が KL1-B 命令列を次々に読みだし, 解釈実行することにより, KL1 プログラムの実行が進む.

5.2　構造体中の未定義変数の扱い

PDSS 処理系では, 図 6 の MRB 方式で示すように, 構造体中の未定義変数は未定義セルを別に取り, 構造体にはそれへの参照ポインタを入れている. 構造体中に未定義変数を直接取るとその構造体本体の MRB が白で回収できる時でも, 構造体中の未定義変数を指している他の参照があるかも知れないので回収出来ないことになってしまう. これでは構造体の回収は全くできなくなる. 未定義変数が具体化され, その値を他の参照パスの全てから読み出されるまで構造体の回収を遅らせる方式も考えられるが, 実行時の処理が複雑になる. このためには, ゴールの実行中断, 再開のメカニズムに似た機能が必要になるからである.

[6] PDSS システムでは, 実際にはメモリに割り付けられている.



WAM 方式　　MRB 方式

図 6: 構造体中の未定義セル

5.3 命令セット

筆者らは, 既に KL1 の抽象命令セット[7, 18] を提案した. 本節では, MRB を正しく管理するために新たに導入した命令について説明する. 導入した命令は, 3.3 節で述べた MRB 管理のための命令と, 4 章で述べたガーベジセル回収用命令である. また, 従来からの命令で, デリファレンス処理を伴う命令については, 3.3.3 節で示したように MRB を管理し, 4.1 節で示したように不要な間接セルの回収を行うように変更した.

5.3.1 MRB 管理のための命令

3.3 節で述べた MRB の更新操作のための命令は, 主としてボディ部でデータへの参照数が増える場合に, そのデータへのポインタの MRB を黒にすることである.

表 2 において, (1) ~ (5) は, 各々, レジスタ Xi が持つデータをレジスタ Aj, ゴールレコードの引数スロットの Gj 番目, リスト L の 'Car' 部, 'Cdr' 部, ベクタ V の I 番目要素位置に転送すると共に, 両者の MRB を黒くする命令である. (6) ~ (10) は, 新たに未定義変数セルを割り付け, オペランドとして与えられたレジスタや, 構造体の中から MRB 黒で指させるための命令である. (11) は, ベクタ V の I 番目要素位置の MRB を黒くする命令である.

表 2: MRB の更新操作のための命令

	命令
(1)	put_marked_value Xi, Aj
(2)	set_marked_value Xi, Gj
(3)	write_marked_car_value L, Xi
(4)	write_marked_cdr_value L, Xi
(5)	write_marked_element_value V, I, Xi
(6)	put_marked_variable Xi, Aj
(7)	set_marked_variable Xi, Gj
(8)	write_marked_car_variable L, Xi
(9)	write_marked_cdr_variable L, Xi
(10)	write_marked_element_variable V, I, Xi
(11)	mark_element V, I

5.3.2 回収のための命令

4 章で述べたガーベジセル回収のための命令である．なお，この命令セットは，デリファレンス命令を陽に持っていないので，デレファレンス時の回収は個々の命令中で動的に行われる．

- collect_list Ai

  レジスタ Ai から指されるリストの MRB が白であれば，リストセルを回収し，フリーリストに繋ぐ．

- collect_vector Ai, Nv

  レジスタ Ai から指される長さ Nv のベクタの MRB が白であれば，ベクタセルを回収し，フリーリストに繋ぐ．

- collect_value Ai

  レジスタ Ai から指されるデータの MRB が白である限り再帰的に回収し，各々の長さに応じたフリーリストに繋ぐ．

- put_reused_list Ai, Aj

  レジスタ Ai から指されるリストの MRB が白であれば，そのリストセルへのポインタをレジスタ Aj に転送する．Aj の MRB は白である．これは，一つのクローズ内で 'collect_list Ai' と 'put_list Aj' 命令が現れ，Ai の MRB が白のとき，一旦フリーリストにリストセルを返さずにそのまま再利用するための最適化の命令である．Ai の MRB が黒であれば，Aj のための新たなリストセルの割り付けが起こる．

- put_reused_vector Ai, Aj, Nv

  これは，Ai が長さ Nv のベクタを指していることを除けば，操作は 'put_reused_list' 命令の場合と同様である．

5.3.3 コンパイル例

MRB-GC 用命令を生成するコンパイル例を図 7に示す．第一引数のリストの 'Cdr' 部 (変数 'Y') 及び第二引数 (変数 'X') に対しては，ボディ部で使われていないので，回収命令が生成される．一方，第一引数のリストの 'Car' 部 (変数 'X') は，ボディ部で 2 回使われるのでマーク命令 (write_marked_car_value) が生成される．さらに，ボディ部でリストが使われているので，第一引数のリストの再利用を試みるための命令 (put_reused_list) が生成される．

5.4 一括型 GC

MRB による GC では，一旦黒くなったデータは後で参照が無くなっても回収できない．従って，別途一括型の GC が必要となる．PDSS 処理系では，このためにコピー方式による GC を備えている [12]．このコピー方式 GC は，旧領域から新領域に生きているデータをコピーする時に，間接ポインタをデリファレンスすることと，参照数が '1' のデータに対しては，一括 GC の終了後には MRB 白のポインタで指されるように MRB の付けかえを行うことに特徴を持っている．

6 実験と評価

実験は，5 節で述べた PDSS 処理系を用いて行った．ベンチマークプログラムとしては，表 3 に示すプログラムを使った．表 3 では，個々のプログラムのリダクション数と機能を簡

```
p([X|Y], X, Z) :- true | Z = [X|ZZ], q(X, ZZ).
```

```
p/3: wait_list  A1                      % 第一引数はリスト?
     read_car   A1, X3                  % car を読む
     read_cdr   A1, X4                  % cdr を読む
     wait_value X3, A2                  % 第二引数の単一化
     collect_value A2                   % 第二引数の回収
     collect_value X4                   % ボイド変数の回収
     put_reused_list A1, X5             % リストの再利用
     write_marked_car_value A1, X3      % 変数'X'の転送とマーク
     write_cdr_variable     A1, A2      % 変数'ZZ'の割付
     get_list_value  X5, A3             % Z = [X|ZZ]
     put_value  X3, A1
     execute    q/1
```

図 7: KL1 プログラムとコンパイル例

表 3: ベンチマークプログラム

プログラム	リダクション	機能
prime	5,876	素数生成
queen	38,878	エイトクイーン
qlay	19,419	レイヤード法[8] による エイトクイーン
bup	34,857	ボトムアップパーサ
kl1cmp	14,919	KL1 で記述した KL1 コンパイラ
espascal	335,115	パスカルの三角形による 係数の計算 (E.Tick[10] による)
etsmall	918,520	詰め込みパズル (同上)
tri	666,235	パズル問題 (同上)
semi	292,309	半群の要素の計算 (同上)
pax	17,530	並列自然言語パーサ

単に示す.

6.1 命令コードの特性

表 4 に, 表 3 の 10 個のプログラムについて MRB-GC を行う場合と行わない場合のコンパイルコードの静的および実行命令数の比 ('MRB-GC を行う場合 /MRB-GC を行わない場合') の平均と標準偏差を示す. この比が, MRB-GC を行うことによる命令数の増加を示す. 表より, MRB-GC を行うことによる静的命令数, 実行命令数の増加は, 各々 5 %, 7 % であり, 全体への影響は大きくないと言える.

表 4: 静的, 実行命令数の比

	平均	標準偏差
静的命令数比	1.05	0.028
実行命令数比	1.07	0.041

6.2 使用ヒープ量

表 5 に, 各々のプログラムを実行するのに必要としたヒープ量 (変数および構造体のための領域) を語[7]数で示す. ヒープは, 'MRB-GC あり' ではフリーリストにより管理され, 'MRB-GC なし' では連続領域を端から使って行く. また, MRB-GC なしでは, 構造体中の未定義変数を構造体の外側に採る必要がないので, 図 6 の WAM 方式のように扱っている[8].

表より, MRB-GC を行うと MRB-GC を行わない場合に比べて, 多くのプログラムで少ないヒープ量で実行できることが分かる. これは, KL1 におけるデータの参照数は多くが '1' であることを実証し, MRB-GC の有効性を示すものである. ただし, 'qlay' の場合には, MRB-GC を行ったほうがメモリを多く消費する. これは, 構造体の変数セルを構造体の外に採ること (図 6) の増分に相当する. このプログラムではネストの深いリストを多く扱うが, ネストの段数分だけ余分に間接セルを割り付けることによるヒープの使用増が顕著に表われた場合と言える. また, 'kl1cmp*' とあるのは, 4.4 節で述べた, 単一参照 (MRB が白) のベクタを再利用しない時の 'kl1cmp' の使用ヒープ量である. KL1 コンパイラでは, レジスタ割り付けの際, コンパイル対象のプログラムに現れる変数の生存区間を管理するためのテーブルをベクタで表現している. このテーブルは, レジスタを最適に割り付けるために何度も参照, 更新される. 従って, このテーブルの再利用を考えず, 破壊的な更新が出来ないことにすると, 更新毎にベクタ本体をコピーする必要が生じる. この場合では, 7 倍近くのヒープを必要とすることがわかり, ベクタ本体の再利用の効果が大きいことがわかる.

[7]一語は MRB を含めたタグ部 1 バイト, データ部 4 バイトから構成される.

[8]PDSS 処理系では, 実際にはコピー式一括 GC の実現を容易にするために, MRB-GC なしでも構造体中の未定義変数を構造体の外側に採っている. この場合, 表 5の全のプログラムで MRB-GC ありが有利となる.

表 5: 使用ヒープ量

プログラム	MRB-GC		比
	なし (A)	あり (B)	(B/A)
prime	10,848	1,503	0.14
queen	817,070	459,919	0.56
qlay	582,146	679,557	1.17
bup	66,655	20,076	0.30
kl1cmp	34,969	17,873	0.51
kl1cmp*	231,797	216,312	0.93
espascal	471,760	50,284	0.11
etsmall	3,174,541	671,354	0.21
tri	1,209,532	1,209,529	1.00
semi	493,383	101,265	0.21
pax	24,298	12,784	0.53

6.3 回収されるデータの生存期間

MRB-GC によって回収されるデータの生存期間を調べるために，プログラムの実行開始から終了までに亘って，変数，リストなどのためにヒープ領域が割り付けられてから，回収されるまでの期間(ライフタイム)をリダクション数で数えた．図 8 に結果を示す．図で，横軸はリダクション数で，縦軸が各リダクション間データが生きた後，MRB-GC によって回収されたヒープ領域の，総回収量に対する割合である．図から，MRB-GC によって回収されるデータの多くは，その生存期間が非常に短く，高々 10 リダクション程度以内に大多数のデータは回収されていることが分かる．実験に用いたプログラムの 1 リダクションあたりに実行される命令数の平均は 20 命令程度であるので，これは，200 命令に相当する．MRB-GC を行わない場合には，変数，リストなどはヒープを単調に消費して割り付けられるので，キャッシュのミスヒットの増大を招くと予想される．一方，MRB-GC を行うと，データの生存区間がたとえ長くても，データが回収されフリーリストに繋がれた後，すぐに使われればキャッシュのヒット率の向上が望める[9]．さらに，この実験データから，一旦割り付けられたヒープ領域は，200 命令程度を実行中に回収され，次のリダクションで再び使われる．従って，これらのヒープ領域は，頻繁に割り付け，開放が起こっていると考えられ，キャッシュ機構を持つプロセッサでは，ヒット率向上に寄与するものと期待できる．

6.4 命令別の回収効率

ヒープの回収は，5.3.2 節で示した命令およびデリファレンス時に行われる．図 9 に，命令別の回収量の内訳を示す．ここでは，'Reuse' 命令 (5.3.2 節) による再利用も一旦回収された後，再び割り付けられたものとして扱った．'Prime' では，リストを使ったストリーム通信を行っているため，'collect_list' による回収が多い．'Queen' では，'collect_value' による回収が多い．これは，探索木の終端クローズのヘッド引数の多くがボイド変数で受けていることによるものと考えられる．また，'qlay' や 'tri' では，殆どがデリファレンス時の回収である．これは，表 6.2 から分かる様に，元々この二つのプログラムは回収率が余り良くなく，僅かに回収された領域がデリファレンスによるものだったことを示している．さらに，どのプログラムでもデリファレンス時の回収の比率が比較的大きい．デリファレンス時の回収は実

[9]多くの場合，回収時にはそのデータは読まれているため．

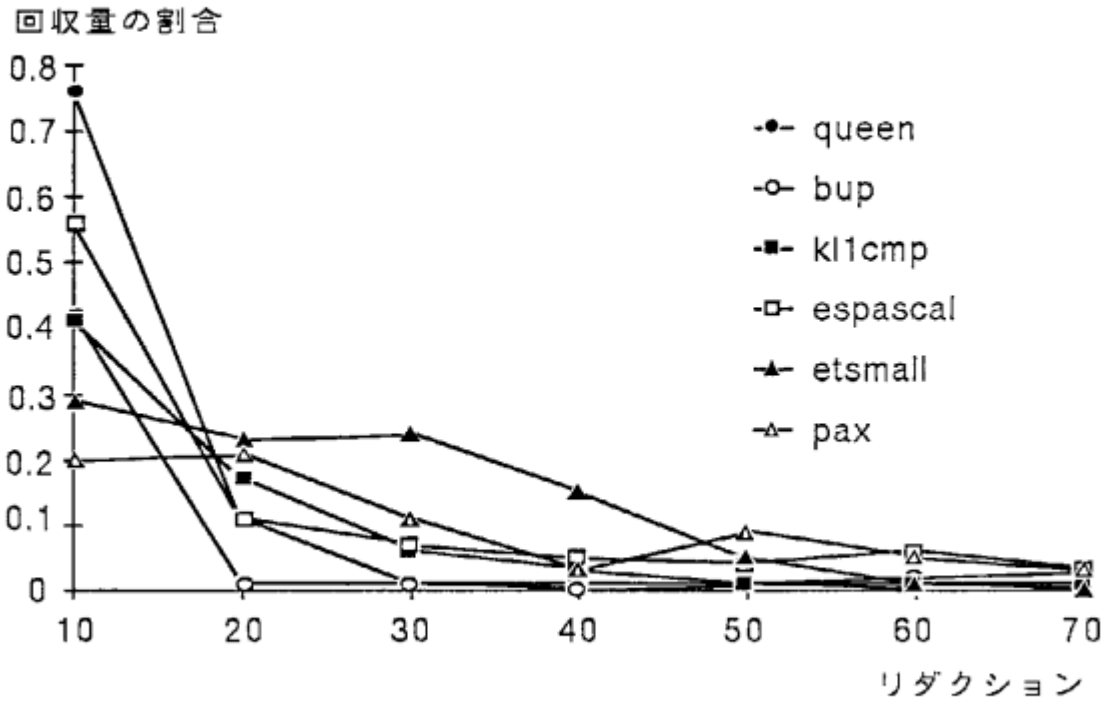


図 8: 回収されるデータの生存期間

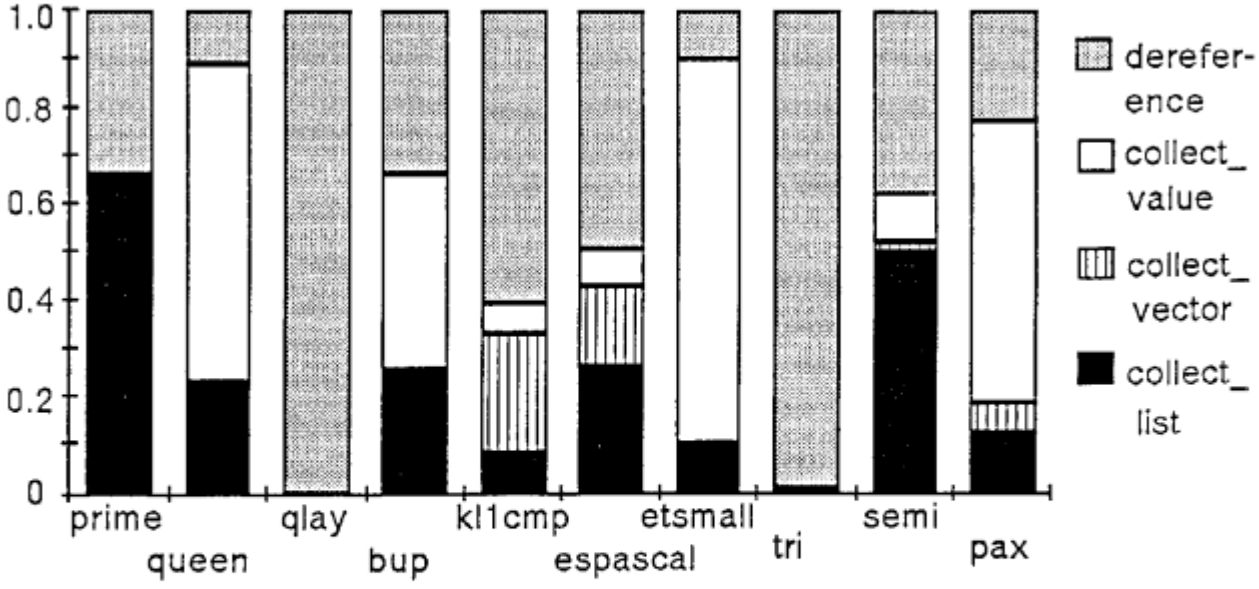


図 9: 命令別の回収効率

行時に動的に行われるため，実際のマシンでの本方式の効率的な実現のためにはハードウエアによるサポートが有効である[9]．

6.5 Reuse 命令の効果

MRB による GC では，メモリ領域はフリーリストにより管理されている．従って，不要になったヒープの回収，新たなヒープの割り付けなどフリーリスト先頭に対する読み，書きの処理が集中すると予想される．そこで，同じクローズ内での同じ大きさの領域が MRB-GC により回収されることが予想され，またボディ部での割り付けが起こることがわかる時には，フリーリストに一旦返すことをせずに，そのまま使ってしまうことが考えられる．この結果，フリーリストへ戻す操作と，新たに取り出す操作を省略できることになり，速度向上はもとより，メモリアクセスの競合をさける上で非常に効果的である．このために導入された命令が 'Reuse' 命令であり，コンスセルおよびベクタ本体について 'Reuse' 命令が生成されている (5.3.2 節)．表 6 に，'prime'，'bup' ，'semi' の三つのプログラムについて，'Reuse' 命令を生成しない場合と，する場合のフリーリストへのアクセス回数を示す．この三つのプログラムでは，全体のフリーリストのアクセスの各々，45 %，20 %，36 % を節約できた．その他のプログラムについては，構造体を 'Reuse' 出来ることが殆ど無く，効果は僅かであった．この最適化は，プログラミングスタイルに依存し，'Reuse' 出来る場合 / 出来ない場合がプログラムによってはっきりしている傾向にある．しかし，コンパイラのみの最適化で実現可能で，'Reuse' 出来ない場合の実行時オーバヘッドは小さく[10]，'Reuse' 出来る場合の効果は大きい．従って，この最適化の実用上の価値は大きいと言える．

6.6 一括型 GC の回数

MRB-GC はインクリメンタル GC の一種であるが，一旦黒くなったデータは回収できないため別途一括型 GC が必要になる．表 7 に，MRB-GC を行う場合と行わない場合の一括型 GC の回数を示す．表で，'メモリ量' は，変数や構造体を割り付けるための領域とゴールレコードなど制御データを置く領域の合計で，大きさは，プログラムを実行できる最小限に近い値に設定した．MRB-GC を行った場合には一括型 GC が起動される回数が減少してい

[10] 'Reuse' しようとする構造体を指すポインタの MRB の白 / 黒を調べるだけである．

表 6: フリーリストアクセス数

プログラム	Reuse 命令		比
	なし (A)	あり (B)	(B/A)
prime	21,497	11,751	0.55
bup	95,257	75,971	0.80
semi	1,056,655	675,045	0.64

ることが分かる．これは，一括型 GC の回数をできるだけ減らすと言う初期の目的を達成するものである．

6.7 アクティブセルと実行時間の関係

MRB-GC を行うと，インクリメンタルなメモリ回収のために実行時の時間が増え，また，MRB-GC を行わないと，一括型 GC の回数が増えるため，一括 GC 時のデータのコピーのための時間が増える．コピー式一括型 GC の時間は，GC 起動時のアクティブセル数に比例する．そこで，一括型 GC の時間も含めた場合の全体の実行時間の比 (‘MRB-GC ありの場合の実行時間’ / ‘MRB-GC なしの場合の実行時間’) を，メモリの総量に対するアクティブセルの割合を変化させて測定した[11]．結果を図 10 に示す．図で，横軸がアクティブセルの割合，縦軸が時間比である．MRB-GC を行う場合では，メモリはフリーリストにより管理されているため，データの割り付け，回収時のフリーリスト管理の時間も含んでいる．‘Prime’, ‘tri’ や ‘espascal’ ではアクティブセルの割合を大きくするために総メモリ量を極端に小さくすると，総メモリ量がプログラム実行時に一時に必要とする最大メモリ量より小さくなり，実行を続行できなくなることが起きたために，アクティブセルの割合が大のときのデータが採れていない．このようなアクティブセルの割合が 10 % 以下と非常に少ないプログラムの場合には，一括型 GC の時間が僅かで MRB-GC を行うことによる時間の方が顕著になる．‘Bup’, ‘qlay’ や ‘semi’ では，緩やかな右下がりのグラフとなっていることから，アクティブセルの割合が大きくなる (‘プログラムの大きさ’ に対してメモリサイズが小さくなる) ほど，

[11] 具体的には，総メモリ量を変化させ，一括型 GC 時にコピーされたデータ量をアクティブセル量として測定した．

表 7: 一括型 GC の回数

プログラム	メモリ量	MRB-GC	
	(Kwords)	なし	あり
prime	50	52	0
queen	20	58	27
qlay	400	3	2
bup	20	8	1
kl1cmp	10	5	2
kl1cmp*	10	29	2
espascal	20	34	2
etsmall	50	72	30
tri	100	10	10
semi	100	19	2
pax	30	1	0

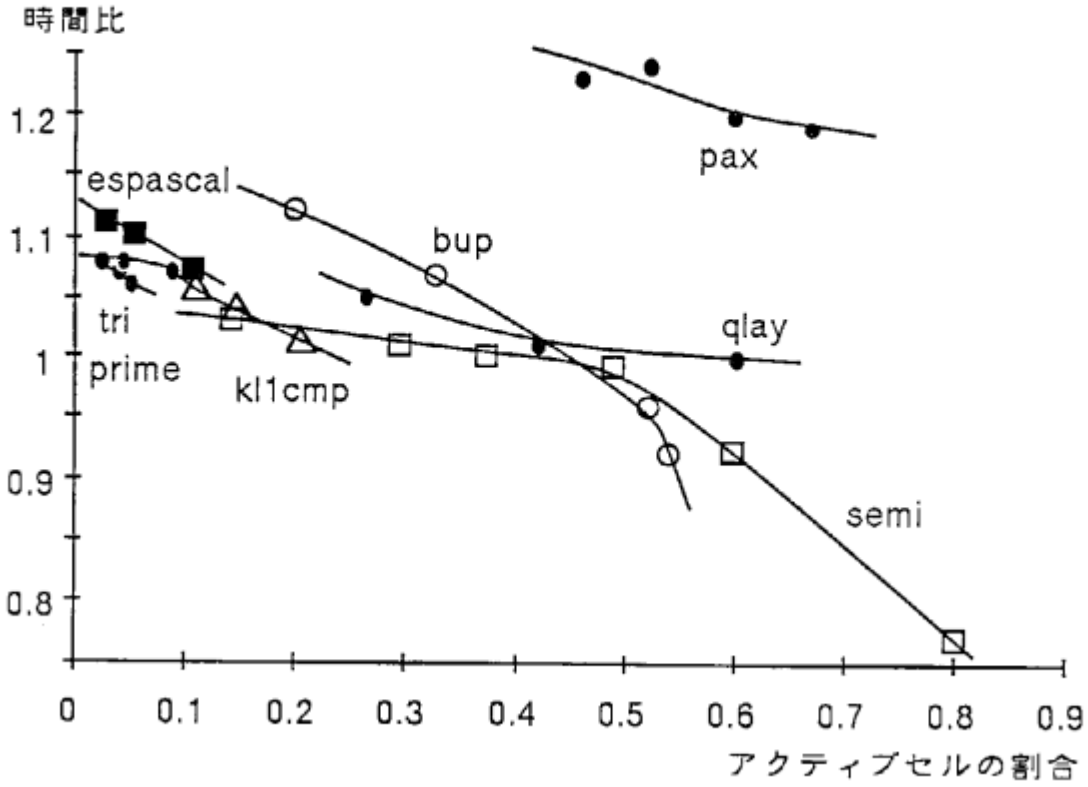


図 10: アクティブセルと実行時間の関係

MRB-GC を行うことによる効果が大きくなっていると言える. アクティブセルの割合が 40 ～ 50 % を越えると MRB-GC を行うほうが実行時間が速くなる傾向にある. 本論文での評価では, C 言語で書かれた汎用計算機上のエミュレータ (PDSS 処理系) を使っている. エミュレータでは, MRB の管理も命令としてエミュレートされるので, 比較的低速である. 一方, 一括型 GC は, C 言語で直接記述されている. 従って, 図 10 では, MRB 管理の時間が一括型 GC の時間に対して大きく出ている. また, 一括型 GC 時にはキャッシュのミスヒットが多くなるが, MRB-GC では, 6.3 節で示したように, キャッシュのヒット率の向上が望める. さらに, 実際のマルチプロセッサ環境では, アクティブセルの割合が増える傾向にあることが報告されている [15]. これらの点を考慮に入れれば, MRB-GC を MRB 管理のためのハードウエアを持つ並列計算機上に実現するならば, この実行時間の増大は小さくなり, 結果として, アクティブセルの割合がそれほど大きくなくても MRB-GC の方が有利になることが予想される. 従って, MRB-GC を実際の並列計算機上に実現することは十分実用性があるといえる.

7 おわりに

本論文では，データ自体ではなく，データへのポインタに参照数が '1' か ' 多数 ' かを表すビットを設け，このビットを管理することにより，参照が無くなった時点で積極的にデータを回収するインクリメンタル GC 方式 (MRB-GC) を提案し，評価を行った．その結果，以下のことが分かった．

1. MRB-GC は，これを行わない場合に比べて，コード量の増加も僅かで，多くの場合，ヒープ量は半分程度でプログラムを実行できる (6.1 節, 6.2 節)．
2. 回収されるデータの生存期間は非常に短くキャッシュ機構を持つプロセッサでは，ヒット率の向上も望める (6.3 節)．
3. 回収命令別による回収データ量の割合では，デリファレンス時のものが比較的多い．デリファレンス時の回収は動的に行われるため，効率的な実現のためには，ハードウエアによるサポートが有効である (6.4 節)．
4. クローズ内で，回収，割り付けが起こる同じ大きさの構造体を再利用することによって，フリーリストへのアクセス回数を減らすことができる (6.5 節)．
5. 一括型 GC の回数を大幅に減らせ，実行時間についても全体のメモリ量に対してアクティブセルの割合が大きいほど，MRB-GC が有利であることが分かった (6.6 節, 6.7 節)．

筆者らは，現在 MRB-GC を支援するハードウエア機能を備えた並列推論マシン PIM/p[9] の開発を進めており，PIM/p のマシンアーキテクチャに合った命令セットの改良などを行っている．MRB-GC を行うことの時間的オーバヘッド，MRB-GC が有利となるアクティブセルの割合，などについては，PIM/p 上で改めて測定する必要がある．これらは，今後の検討課題である．

謝辞

日頃御指導頂く ICOT 第 4 研究室，内田俊一室長に感謝します．また，PDSS 処理系を使って実験を行ってくれた，富士通 SSL の武井則雄氏，平野喜芳氏，貴重なコメントを頂いた ICOT 第 4 研究室，後藤厚宏氏をはじめとする ICOT 及び関連メーカの方々， PIM/MPSI

ワーキンググループ, PIM/MPSI開発グループ, 並列処理検討会のメンバの方々に感謝します.

参考文献

1) Baker,H.G.: List Processing in Real Time on a Serial Computer. *Comm. ACM*, Vol.21, No.4, pp.280–294, 1978.

2) Chikayama,T., Kimura,Y.: Multiple Reference Management in Flat GHC. Proc. *Int. Conf. on Logic Prog. '87*, pp.276–293, 1987.

3) Chikayama,T., Sato,H., Miyazaki,T.: Overview of the Parallel Inference Machine Operating System (PIMOS). Proc. *Int. Conf. on FGCS '88*, pp.230–251, 1988.

4) Cohen,J.: Garbage Collection of Linked Data Structures. *ACM Computing Surveys*, Vol.13, No.3, pp.341–367, 1981.

5) Deutsch,L.P., Bobrow,D.G.: An Efficient, Incremental, Automatic Garbage Collector. *Comm. ACM*, Vol.19, No.9, pp.522–526, 1976.

6) ICOT 第四研究室: PDSS - 言語仕様と使用手引き -. TM-437, ICOT, 1988.

7) Kimura,Y., Chikayama,T.: An Abstract KL1 Machine and its Instruction Set. Proc. *Symp. on Logic Prog. '87*, pp.468–477, 1987.

8) Okumura,A., Matsumoto,Y.: Parallel Programming with Layered Streams. Proc. *Symp. on Logic Prog. '87*, pp.224–231, 1987.

9) Shinogi,T., et. al: Macro-call Instruction for the Efficient KL1 Implementation on PIM. Proc. *Int. Conf. on FGCS '88*, pp.953–961, 1988.

10) Tick,E.: Performance of Parallel Logic Programming Architectures. TR-421, ICOT, 1988.

11) Ueda,K.: Guarded Horn Clauses : A parallel logic programming language with the concept of a guard. TR-208, ICOT, 1986.

12) 宮内, 木村, 近山, 久門: MRB による多重参照管理方式 - KL1 処理系における一括型 GC の特性評価 -. 第 35 回情処全大 (昭和 62 年後期), 2Q-7, 1987.

13) 近山, 木村: ポインタタグ (MRB) による多重参照管理方式. 第 35 回情処全大 (昭和 62 年後期), 2Q-5, 1987.

14) 今井, 木村, 稲村, 後藤: 並列推論マシン PIM における効率的構造体処理方式 - 参照数管理と長男優先方式 -. 信学技報 CPSY88-47, 1988.

15) 佐藤, 後藤: KL1 並列処理系の評価 - メモリ消費特性と GC -. Proc. 並列処理シンポジウム *JSPP '89*, pp.195-202, 1989.

16) 小沢, 林, 服部: 実時間 GC の実現方式と評価. 情処論, Vol.29, No.5, pp.465-471, 1988.

17) 木村, 近山: MRB による多重参照管理方式 - KL1 処理系における実時間ガーベジコレクション方式 -. 第 35 回情処全大 (昭和 62 年後期), 2Q-6, 1987.

18) 木村, 近山, 久門, 中島 (浩): 並列推論マシン PIM - KL1 の抽象命令仕様とコンパイラ -. 第 34 回情処全大 (昭和 62 年前期), 2P-1, 1987.

19) 木村, 西田, 宮内, 近山: KL1 の多重参照ビットによる GC 方式について. Proc. データフローワークショップ *'87*, pp.215-222, 1987.