記号処理 18－9
（1982. 3. 25）

高速リスト処理に適したアーキテクチャについて

服部 彰　篠木 剛　品川 明雄　林 弘
（富士通研究所）

1．はじめに

人工知能の研究等に広く使われているＬＩＳＰ言語には、記憶域の動的割付、関数の再帰呼出、リスト処理など効率的に処理するにはアーキテクチャや方式的なサポートを必要とする処理が多い。

実用的なＬＩＳＰ専用機としては以下のような機能が必要である。

（1）高速大容量のスタック
- 特に、内部レジスタ並みのアクセス速度をもつていること。
- 関数を能率よく実行するために各種ポインタを備えていること。
- マルチプロセスを能率よくサポートできること。

（2）高速大容量のセル空間
- 実用的には仮想記憶サポートが必要
- 大容量セル空間でも効率のよいガーベジコレクタ

（3）強力な条件分岐機能
- データタイプによる分岐など

（4）コンパイルした関数を高速実行する機能
- 命令の先取り機構や命令デコード、ディスパッチ機構

（5）インタープリタや基本関数のファームウェア化
- 大容量の高速制御記憶

ここでは特に、マルチプロセスを効率よくサポートできる高速スタック方式と、大容量のセル空間でも効率よく実時間処理ができるガーベジコレクト方式について報告する。

2．仮想マルチスタック

ＬＩＳＰではスタックは極めて[1]頻繁に使用されるため、ＬＩＳＰ専用マシンでは通常高速メモリ上に実現される。

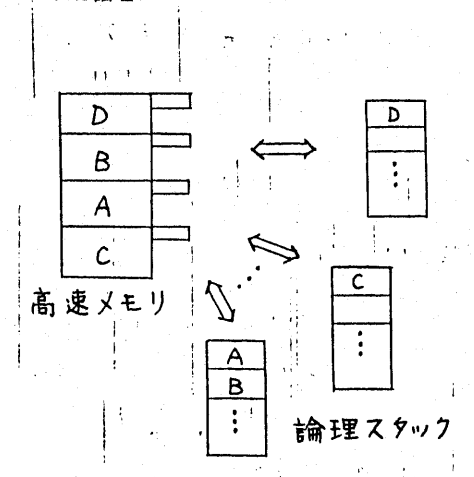
一方、マルチプロセスの場合には各プロセス毎にスタックが必要であり、変数のＢＩＮＤ法としては各プロセスの環境（Ａ－ｌｉｓｔ）を別々に管理しやすいＤｅｅｐ－Ｂｉｎｄ法が適している。そのかわり、スタックの大きさを十分にとる必要がある。しかし、これらのスタック群を全て高速メモリ上に置くことはコスト的に問題である。そこで、スタック用の高速メモリを各プロセス間で共用する「仮想マルチスタック方式」について検討した。

2．1　基本構成

高速メモリをプロセス間で共用する場合、高速メモリ全体を現在実行中のプロセスに割付けてしまうと、プロセスを切り変える時のスタックのスワッピングのためのオーバヘッドが大きい。そこで、各プロセスの仮想スタックと高速メモリ（物理スタックとよぶ）をブロックに分割し（各々、論理ブロック、物理ブロックとよぶ）、高速メモリをブロック単位でプロセスへ割付ける方法を採る。そのため、物理スタックの各ブロックには、割付けられている仮想スタックと論理ブロック番号を識別する情報を格納するタグ（ディレクトリ）を設ける。（図．１参照）

2．2　アドレス変換

プロセスは論理アドレスでスタックをアクセスするため、



図．１　仮想マルチスタック

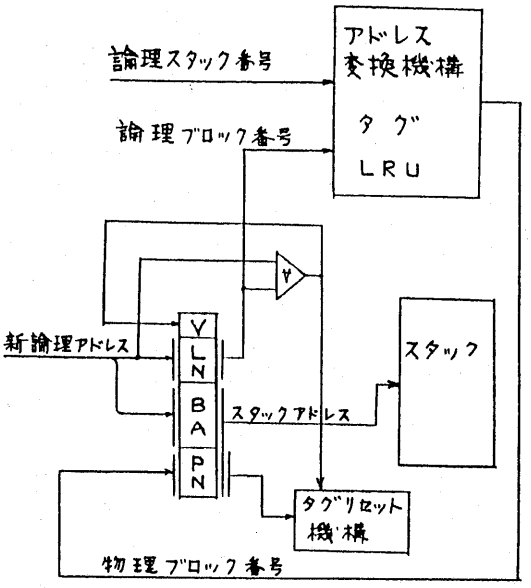
物理スタック上の実アドレスへの変換機構が必要である。しかし、通常のキャッシュのようにアクセス毎にタグの検索を行うのは、レジスタ並みの高速性が要求されるスタックにおいては適当ではでい。スタックにはその先端部分が頻繁に使われるという局所性がある。そこで、スタックポインタがブロック境界を越えた時のみ、その論理ブロック番号をタグ（NOT－FOUNDの時にはLRU回路）により物理ブロック番号へ変換してポインタへ格納する。これにより、論理アドレスが変化してもブロックが変らない限り、ポインタ中の物理ブロック番号とブロック内アドレスが実アドレスになり、通常のスタックアクセスではタグの検索を必要としない。（図．2参照）

論理アドレス＝論理ブロック番号||ブロック内アドレス

物理アドレス＝物理ブロック番号||ブロック内アドレス

図．2のアドレス変換機構において、タグは、各物理ブロック毎に、割付けられている仮想スタック識別情報、論理ブロック番号及び使用中表示フラッグを保持している。

一方、スタックポインタは論理ブロック番号（LN）、ブロック内アドレス（BA）、物理ブロック番号（PN）、有効表示ビット（V）から構成されている。



図．2　ポインタのアドレス変換

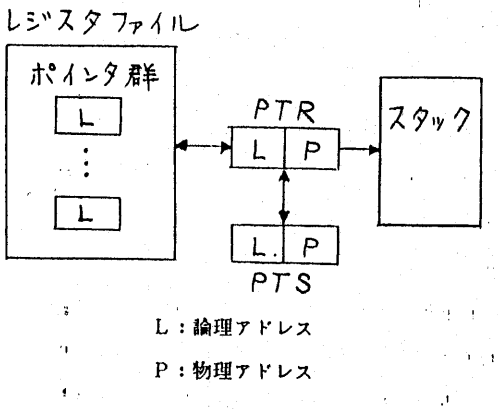
2．3　アクセス制御とポインタ

ＬＩＳＰを効率よく実行するためには、スタックの先端を指すスタックトップポインタ（ＳＴＰと略す）以外に、環境の維持、変数の高速アクセス等のために各種のポインタが必要である。これらのポインタ群は論理スタック上で局所性があり、同一ブロックを指している場合が多い。

そこで、仮想スタックアクセスに関する制御機構を備えたポインタレジスタＰＴＲとその退避用レジスタＰＴＳをこれらポインタ群で共用する方式を採用した。（図．3）

ＰＴＲは、図．2のように、ブロック境界を越えたことを検出するために、ＰＴＲへ新しいポインタを転送したり増減した場合に、論理ブロック番号の新旧値を比較し、一致しなければＰＴＲの有効表示ビットをリセットする機能をもっている。そして、スタックをアクセスした時、ＰＴＲの有効表示ビットが落ちていると、スタックフォールト割込みが発生し、その割込み処理の中で図．2のアドレス変換機構を用いて、新物理ブロック番号をＰＴＲへ設定する。この時、必要ならばブロックのスワッピング処理を行う。又、ＰＴＲをＳＴＰとして使ってスタックが縮む方向へブロック境界を越えた時には、タグリセット機構がもと居たブロックのタグの使用表示フラッグをリセットする。

以上の「仮想マルチスタック」方式により、マルチプロセスのサポートに不可欠な高速大容量のスタック群を提供することができる。



図．3　ポインタレジスタの共用

3．実時間ガーベジコレクタ

ここでは、大容量メモリ特に仮想記憶に適した、効率のよい実時間ガーベジコレクタについて述べる。

並列ガーベジコレクションの方法は大きく分けて、2つのタイプに分類することができる。1つは、基本的にＴｒａｃｅ&Ｍａｒｋ法を用いるタイプであり、他の1つは参照カウンタを用いるタイプである。

第1のタイプの場合、例えばＳｔｅｅｌe[2]が詳細なアルゴリズムを記述しているが、リスト処理プロセスとＧＣプロセスの間で各種資源のアクセスの度にセマフォア操作が多用されている。特に、スタックをアクセスする時にも同期処理が必要であり、スタックの使用頻度が高いため、リスト処理プロセスのオーバヘッドは非常に大きい。更に、仮想記憶の場合、二次記憶上にあるページについても全てトレースしなければ回収処理ができないため、回収時間が非常に長くなる。

第2のタイプの場合、その欠点[4]として（1）参照カウンタのために余分なスペースが必要なこと、（2）参照カウンタを更新するためのオーバヘッド、（3）一般的に循環リストを回収できないこと、が知られている。しかし、回収の時に全空間をトレースする必要がないため、仮想記憶の場合には有利である。

Ｄｅｕｔｓｃｈ[3]らは上記の欠点を軽減する方法を提案したが、依然としてリストを作成、変更する時に管理テーブルを更新したり、検索する大きなオーバヘッドが存在する。

参照カウンタを用いる方法のこれらのオーバヘッドは、基本的に各セル単位で回収できるかどうかを判断するために参照カウンタを用いていることからきている。

そこで、我々はセル空間を部分空間に分割し、参照カウンタを各セルの回収の可否を判断するのに用いるのではなく、部分空間単位の回収に用いる方法を提案する。つまり、部分空間の外部から直接参照されるセルを識別するために、参照カウンタを用いている。

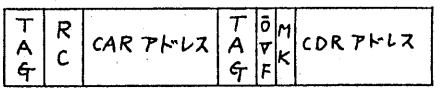
3．1　基本的方法

セル空間は部分空間（例えば、ページングシステムのページ）に分割され、各セルは図．4のようにＣＡＲ、ＣＤＲ域以外に参照カウンタ域（以下、ＲＣと略す）をもっているものとする。ＯＶＦフラグはＲＣ域が溢れたことを示し、それ以降のＲＣの変更分は、マルチ参照テーブル[3]（以下、ＭＲＴと略す）というハッシュテーブルへ格納する。

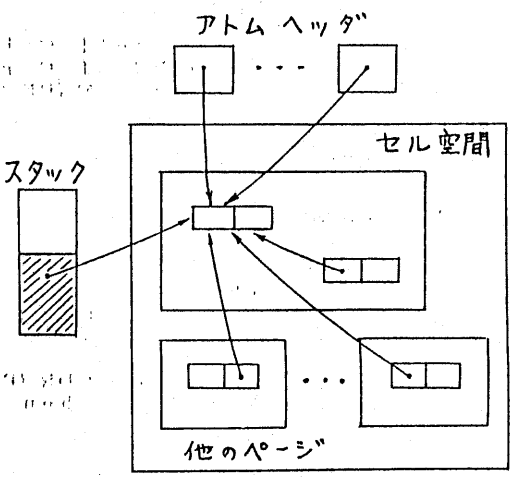
一般に、各セルは図．5のようにスタック語、他のセル、及びアトムヘッダから参照される。もし、これらの参照を全てＲＣに反映させれば、回収処理は簡単になるが、リスト処理プロセスでのＲＣ更新のオーバヘッドが非常に大きくなる。そのため一部の参照はＲＣの対象からはずすことを考えなければならない。

まず、スタック上のデータは頻繁に変化するので、ＲＣ更新のオーバヘッドを減らすために、スタックからの参照はＲＣの対象から除外する。

次に、同一部分空間の中にあるセルからの参照もＲＣの対象から除外する。これにより、リスト構造を作ったり変更したりする場合でも、参照が部分空間の境界を越えていなければ、相手先セルのＲＣを更新する必要がなくなる。



図．4　セルのフォーマット



図．5　セルに対する参照源

一方、他の部分空間の中のセルからの参照はＲＣに反映させる。それは、これらのセルの個数が多いことと、回収処理を部分空間に限定するためである。同様の理由から、アトムヘッダからの参照もＲＣに反映させる。

以上から結論として、リスト処理プロセスのオーバヘッドを減らすために、次のものはＲＣの対象から除外する。

・スタックからの参照

・同一部分空間の中のセルからの参照

一方、回収処理を部分空間に限定できるように、次のものはＲＣの対象に入れる。

・他の部分空間の中のセルからの参照

・アトムヘッダからの参照

以上により、部分空間の中の活動セルは、この部分空間を直接参照しているスタック語、及び非零ＲＣを持つこの部分空間内セルを起点としてトレースし、刻印することができる。そこで、刻印ビットがｏｆｆのセルを不要セルとして、その部分空間のフリーセルリストへつなぐ。

3．2　アルゴリズム

スタックと、その部分空間内の非零ＲＣセルからのマーキング操作は、以下の関数ＳＴＡＣＫＭＡＲＫ　及びＮＺＲＣＭＡＲＫ　のように記述することができる。

回収操作は関数ＲＥＣＬＡＩＭ（次のページ）のように記述できる。その際、回収されるセルが他の部分空間のセルを参照していた場合には、相手先セルのＲＣを－１する。しかし、相手先セルが二次メモリ上にある場合には、関数ＲＣＤＥＣＲＥＭＥＮＴ　のように、テーブルＭＲＴを検索し、相手先セルに対応する項目があればそのＲＣ変更分を－１し、なければ新しい項目を設けＲＣ変更分として－１を登録する。

```
STACKMARK (PAGE) =
   prog (P)
       for each pointer P in stack do
           if P points to the page PAGE
              then  MARK (P)
           fi
       od
   end


NZRCMARK (PAGE) =
   prog (C)
       for each cell C in the page PAGE do
           if  ( RC of C ≠ 0 )  and  ( MK of C = off )
              then  MARK (C)
           fi
       od
   end


MARK (P) =
   prog ( )
       if MK of P = off
           then  MK of P := on ;
                 if CAR (P) points to the cell in PAGE
                    then  MARK (CAR (P) )
                 fi;
                 if CDR (P) points to the cell in PAGE
                    then  MARK (CDR (P) )
                 fi
       fi
   end
```

```
RECLAIM (PAGE) =
  prog (COUNT, C)
      COUNT := 0;
      for each cell C in the PAGE do
          if MK of C = off
            then
                if CAR (C) points out of PAGE
                  then RCDECREMENT (CAR (C))
                fi;
                if CDR (C) points out of PAGE
                  then RCDECREMENT (CDR (C))
                fi;
                append the cell C to the free list in PAGE;
                increment COUNT
            else
                MK of C := off
          fi
      od;
      return COUNT
  end

RCDECREMENT (C) =
prog ( )
if cell C is in the real memory
  then decrement RC of C
  else
      if cell C is in the MRT
        then
            decrement the associated count in the MRT
        else
            enter the cell C in the MRT with a count of -1
      fi
fi
end
```

3．3　この方法の長所

この方法の長所は以下の通りである。

（1）仮想記憶の場合、このＧＣは実メモリ上のページを選んで回収することができるので、ページフォールトを避けることができる。

（2）上記（1）及び、部分空間の回収時間は短いため実時間処理ができる。

（3）リスト構造を作ったり変更する時でも、参照が部分空間の境界を越えなければ、ＲＣを更新する必要がない。これにより、リスト処理プロセスのＲＣ更新オーバヘッドが大幅に軽減される。

（4）部分空間単位に回収処理を完了させることができるため、リスト処理プロセスとＧＣプロセスの間の同期操作が不要である。

（5）マーキング操作をスタック法で行う場合、そのスタック長は高々部分空間の中のセルの個数で済む。

（6）部分空間の中に閉じる循環リストは回収できる。

3．4　このＧＣの使用法

ページングシステムの場合、セル空間の各ページの状態はそれが実メモリ上にあるかどうか、自由セルを持っているかどうかで図．６のように表すことができる。

この状態にもとずいてセルの消費と回収処理をページ単位で行うために、各ページの状態を識別、管理するテーブルを設ける。

ＧＣは、実メモリ上にある自由セルの数が減ってくるか入出力待ち等で動作可能なプロセスが存在しない時に起動される。起動されると、自由セルが尽きた実メモリ上のページの一つを選んで回収処理を行い、一定数以上のセルが回収できれば、図．６のように、そのページを自由セルが存在するグループへ移す。

一方、関数ＣＯＮＳ　は自由セルが存在する実メモリ上のページからセルを取得し、そのページ中のセルが尽きると、そのページを自由セルが存在しないか一定数以下のグループへ移す。

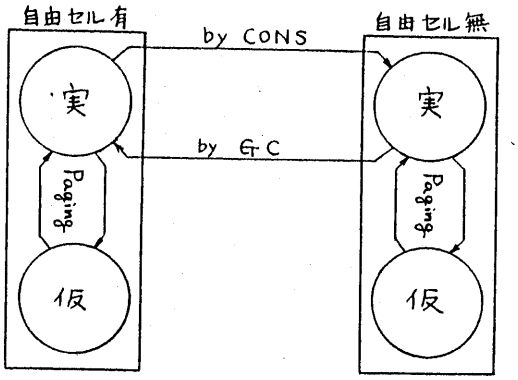


図. 6　セル空間のページの状態遷移

3．5　効率と時間

仮想記憶の場合、二次メモリとのページング動作が回収時間の殆どを占める。

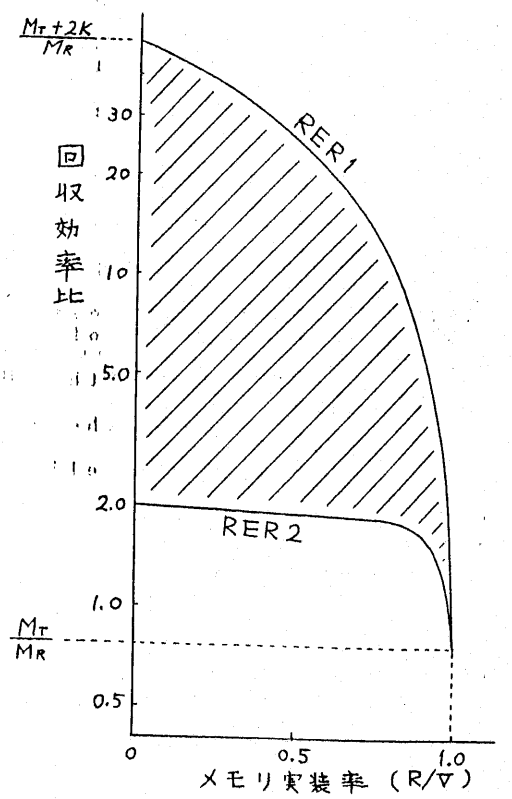
基本的にＴｒａｃｅ＆Ｍａｒｋ法を使うＧＣの場合、刻印操作と回収操作の間に、各ページが少なくとも２回ページインされる。

一方、本方式の場合には部分空間単位に回収できるので、回収の対象となるページを実メモリ上から選ぶことができる。しかも、二次メモリ上のページから回収する場合でも、各ページ当り１回のページインで済む。

ここで、回収効率を単位時間当りに回収されるセルの数と定義する。そうすると、Ｔｒａｃｅ＆Ｍａｒｋ法を用いるＧＣに対するこの部分空間単位ＧＣの回収効率比は図. 7のように表せる。但し、条件は活動セルの割合０．５、主記憶と二次記憶の速度比５０Ｋ、部分空間のサイズ５１２セルの場合である。

図. 7において、ＲＥＲ１は本ＧＣが実メモリ上のページだけで十分なセルが回収できる場合の回収効率比であり、ＲＥＲ２は本ＧＣが二次メモリ上のページからも回収しなければ十分なセルが回収できない場合の回収効率比である。つまり、ＲＥＲ１は活動セルの割合が低い場合の回収効率比の上限を、ＲＥＲ２は活動セルの割合が高い場合の回収効率比の下限を示している。

次に、本方式の部分空間の回収時間をＴｒａｃｅ＆Ｍ－ａｒｋ法を使うＧＣの全空間の回収時間と比較したのが図. 8である。これから、仮想記憶の場合はもちろん、一階層の大容量メモリの場合にも実時間処理に適していることがわかる。



$$RER = \{M_T + 2K(1 - R/V)\} / M_R$$

$$RER = \{M_T + 2K(1 - R/V)\} / \{M_R + K(1 - R/V)\}$$

$K = T/N$

T　：メモリ階層間の速度比

N　：部分空間のサイズ

M_T　：Ｔｒａｃｅ＆Ｍａｒｋ法での
セル当りメモリ参照回数

M_R　：本ＧＣ方式でのセル当りのメモリ参照回数

M_T と M_R は活動セルの割合の関数である。

図. 7　本方式と従来方式の回収効率比