記 号 処 理 50－1
（1989. 3. 14）

Pointer-Linked Data における仮想記憶管理の一手法

前川博俊、°安田弘幸、沢田佳明、福田譲治

ソニー株式会社・総合研究所

Pointer-linked data（以下リストデータ）によるデータ表現手法は、知識処理、自然言語処理等の分野において広く利用されている。近年、これらの分野においては、実用的な大規模システムの開発が進み、大量のリストデータをより効率的に仮想化できる手法の研究が求められている。従来より広く用いられているページングによる仮想記憶管理手法は、空間的および時間的局所性の高いデータに対しては有効性が広く知られているが、一方、局所性の乏しいリストデータに対しては同様の有効性を期待することは困難である。

本論文では、リストデータの性質を利用することにより、従来に比較し高い処理効率を期待できる新たな仮想記憶管理方式—ＳＯＳＯ—を提案し、さらに、同手法を実験的に小型の言語処理系に適用し、ページング手法との比較検討シミュレーションを行い、効率向上の可能性について論じている。

The Virtual Storage Management Method for Pointer-Linked Data

Hirotoshi MAEGAWA, °Hiroyuki YASUDA, Yoshiaki SAWADA and George FUKUDA

Corporate Research Laboratories, Sony Corporation
4-14-1, Asahi-cho, Atsugi, Kanagawa, 243 Japan

The further improvement of virtual storage management method for pointer-linked data is now strongly needed with the increasing interests in development of a number of practical AI systems.

The lack of locality in both time and space domain of pointer-linked data makes paging method difficult to be effectively improved.

This paper describes the new virtual storage management method refered to as SOSO especially designed for pointer-linked data, and then studies the experimental results of computer simulation of SOSO in comparing with paging method.

1. はじめに

我々は、記号処理をより高速にかつ大規模に実行することが可能なマシンの研究を行っている。その一環として、記号処理分野で特に重要なデータ構造であるリストデータにおける仮想記憶の管理方式に関して、従来から広く取り入れられているページング方式[1]に比べて、より高効率を期待できる仮想記憶管理方式 — ＳＯＳＯ (Structure Oriented Storage Organization) — を考案したので報告する。

近年、半導体製造技術やソフトウェア技術などの進歩によって、計算機は高速化、多機能化、かつ小型化してきており、同時に、計算機に実行させる仕事も、大規模化、複雑化してきている。特に、数値計算だけでなく、記号処理などを実行させた場合、汎用の計算機ではパフォーマンスが不足している場合が少なくない。このようなパフォーマンスの不足を補うために、それぞれのアプリケーションに適合したアーキテクチャを設計し、さらに高速多機能化を目指したものが、専用マシンである。専用マシンは、その上で実行されるアプリケーションが比較的限定されているために、それに見合ったハードウェアを構築することが可能であり、汎用マシンと比較すると、コンパクトで高速多機能化が容易である。記号処理の分野においても、この考え方に基づいた種々の専用マシン[2] [3]が発表されている。これらのマシンは、それぞれ、そのアーキテクチャに独自の記号処理向きな工夫が凝らされている。

さて、記号処理向きの専用マシン（以下、記号処理マシンという）を設計する場合、重要な部分となるものの一つにメモリの管理方式が挙げられる。特にリストデータを扱い、２次記憶を用いて記憶空間の仮想化を行う場合、メモリ管理方式の選択はそのマシンのパフォーマンスを決定する重要なファクターである。

本論文では、記号処理マシンの開発にあたり、リストデータに関するメモリ管理方式の検討を行い、ＳＯＳＯと呼ぶ一方式を考案したので報告する。本稿では、最初にＳＯＳＯの概要について、次にその動作の詳細について述べ、最後にシミュレーションによる本方式の評価結果について述べる。

2. 仮想記憶とリストデータ

2.1. ページング方式における記憶領域の仮想化

一般的なワークステーションなどの計算機のメモリを仮想的に拡張する方法として、広く利用されているものに、ページング方式によるものがある。この方式は、図１に示されるように、実記憶をいくつかのページに分け、そのページを単位に２次記憶とデータの交換を行い、記憶領域を仮想的に広げるものである。多くの記号処理マシンにおいても、この考え方が採用され、インプリメントされている[4]。

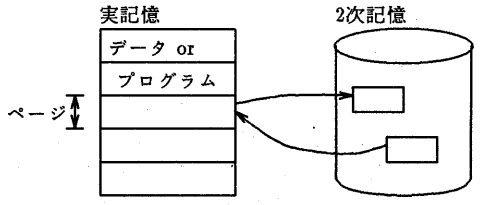


図１．ページングによる仮想記憶方式

一方、大多数の記号処理マシンおいては、内部のデータ構造としてリストデータを用いている。例えば、いまリストデータが、仮想空間上に図２に模式的に示されているような構造で存在していると仮定する。

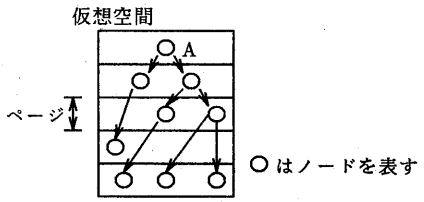


図２．仮想空間上のリストデータ

Ａで示されるリストデータは、各ノードが別々のページに分離している。仮に、これらの内、数ページがページアウトされている状態であれば、１本のリストを手繰る間にこれら複数のページをすべてページインする必要が生じる。この例のように、リストデータは、記憶空間内において空間的な局所性がなく、ページング方式には不向きなデータ構造を形成しているのが一般的である。

また、同一ページには、関連のない複数のリストデータも存在するため、１回のページアウトによってその動作の起因となったリストデータと関係のないリストデータまで同時にページアウトされる可能性が高く、さらに、不要なページインを招く可能性も大きい。この様なページフォルトの頻繁な発

生は、リストデータの処理効率低下の大きな要因である。一方、データ操作の方法やデータ構造を工夫することによって、局所性を持たせることも可能であるが、この場合、データの動的変化には追従しにくいという問題を生じる。ガーベジ・コレクション（以下ＧＣという）の場合に、データ領域の全空間をアクセスする場合が多く、ページフォルトの増加につながる。

以上述べてきたように、一般にページング方式では、１回のページイン／ページアウトに必要とする処理時間は比較的短く、比較的簡単なハードウェアで実現でき、この意味では優れた方式であるが、一方、記号処理マシンに用いた場合には、ページフォルトが頻繁に起こり、システム全体の効率を落とす要因となる場合が少なくない。

今回、以上を考慮し、仮想記憶管理をよりリストデータ向きとした方式を考案した。

2.2. ＳＯＳＯにおける仮想記憶管理方式

本稿で提案しているＳＯＳＯでは、図３に示したように、ページング方式のページに対応する概念としてリスト構造そのものを考え、２次記憶とのデータの転送をリスト構造のままで行おうとするものである。リスト処理中、実記憶のフリーエリアが減少したとき、実記憶上のデータを２次記憶上に転送することをリストアウトとよび、逆に、リストを手繰ったとき、目的のデータが実記憶上になく２次記憶中の該当するデータを実記憶中に転送することをリストインと呼ぶ。また、目的のデータが実記憶上にない状態をリストフォールトと呼ぶ。

本方式では、リストイン／リストアウトに要する処理時間は、ページイン／ページアウトのそれよりも増加する。しかし、リストフォールトの発生回数をページフォールトに比べて大幅に減少させることができれば、システム全体の効率は向上することが期待できる。

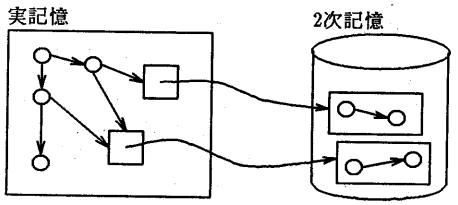


図３．ＳＯＳＯによる仮想記憶方式

3. ＳＯＳＯの概要

3.1. 適用可能なデータ構造

ＳＯＳＯは、以下に示すような性格を持つリストデータを扱う。

(1) 各ノード間がポインタで結合されている。

(2) ポインタは基本的に１方向である。

(3) ノードは他のノードへのポインタと自分自身のデータを持つ。ポインタ数は２以上である。

(4) メモリ空間上では、ノードは番地付けられたメモリ上のデータ領域、ポインタはその領域のアドレスで表現される。

このリストデータの構造は Lisp 等の処理系で一般的に知られているものである。

3.2. リストイン／リストアウトの発生

リストデータが動的に変化してゆく過程においてＳＯＳＯ方式によるリストイン／リストアウトが発生する条件とその動作は、以下のとおりである。

［リストイン］

必要なデータが実記憶上になく、リストフォールトが発生し、そのデータが２次記憶上にあるとき、そのデータを含む一連のリストデータを実記憶上に転送する。

この一連のリストデータをストラクチャと呼ぶ。

［リストアウト］

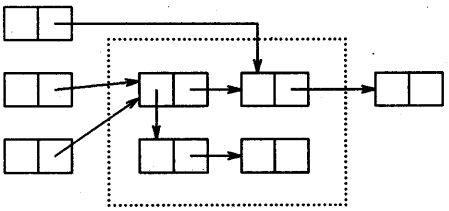
実記憶上でＧＣ実行後、フリーエリアのサイズがある閾値を越えない場合に、リストアウトが起動される。

3.3. 実記憶と２次記憶上での、リストデータの表現

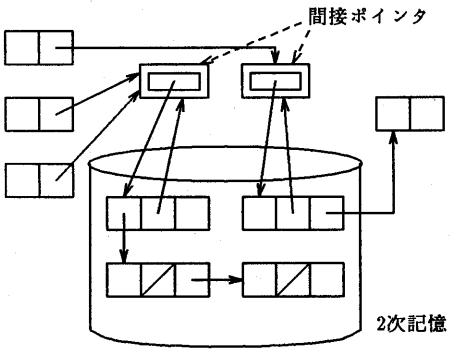
まず、図４－ａに示すリストデータを考える。点線で囲った部分がリストアウトされたとすると、その後の状態は図４－ｂに示すとおりとなる。ＳＯＳＯでは、実記憶内から２次記憶内へのリストデータの参照は、間接ポインタを用いて実現されている。このため、通常はポインタから２次記憶空間を直接指すことはなく、実記憶上のポインタの幅は実記憶空間のサイズのみをアドレッシングできる範囲で良い。従って、２次記憶領域の大きさと実記憶領域の大きさとの間には依存関係がなく、２次記憶を含めたシステム全空間の拡張も容易に行うことができる。この、システム全体の記憶空間の拡張性の高さはＳＯＳＯの利点の一つと考えられる。

また、リストアウトされたストラクチャ内のポインタは、基本的にはその内部で閉じているため

(a) リストアウト前のデータ構造



(b) リストアウト後のデータ構造



□□ はノードを表す。

実記憶内では１ノードは２ポインタである。

２次記憶内では１ノードは３ポインタであり、うち１つは実記憶への逆ポインタである。

図４．ＳＯＳＯにおけるノードの表現

に、２次記憶内のポインタの幅は、実記憶とリストアウトするストラクチャのサイズをアドレッシング出来る範囲で良い。

前述の間接ポインタは、実記憶上に存在し、２次記憶空間を指し示している。従って、間接ポインタは、２次記憶のサイズに応じた幅が必要である。

3.4. リストアウトの対象となるストラクチャの選択

リストアウトするストラクチャの選択で最適なのは、今後アクセスするであろうタイミングが最も遅く、かつその回数が最も少ないストラクチャを選択することである。言い換えれば、時間的、空間的に最も遠いストラクチャを選択することということができる。

リストデータは処理される過程において、図５に示される構造を形成する場合が多い。丸で囲んだ領域はひとつの処理単位(プロセスあるいはタスクなど)におけるデータ環境であるとする。この場合、実行時のある瞬間には、これらの環境のうちのいづれか一つのみに参照が集中していることが多いと考えることができる。このとき、処理系から処理単位の実行状況等の情報を得ることが可能であれば、時間的にもっとも遠い環境を選択することは比較的容易であり、その環境の一部をリストアウトするストラクチャとして選択すれば、処理効率が向上する可能性が高い。

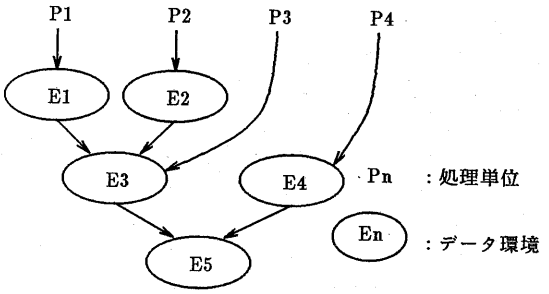


図５．リストデータ処理過程での環境の構造

3.5. リファレンス・カウンタ

リストアウトの際、リストアウトされたリストデータを参照するための間接ポインタを作るか否かを判断するため、各ノードにリファレンス・カウンタを設けている。リストアウトする際、このカウンタの値が、２以上なら今参照したポインタ以外にも参照があることとなり、間接ポインタを作り、これによって仮想的に実記憶上に実体を残しておくことができる。また、リファレンス・カウンタは当然、ＧＣにも利用することが可能である。

4. リストイン／リストアウトの動作

ＳＯＳＯにおける処理の具体例を以下に概説する。

4.1. リストアウト

4.1.1. リストアウトの手順

(1) 実記憶上にフリーノードが存在しなくなると、実記憶上でのＧＣを行う。

(2) ＧＣによるフリーノードの回収の後、フリーノード数が一定の閾値（以後Ｓｈで表す）を越えないときは (3) に進む。回収したフリーノード数がＳｈ以上であれば、リストアウトは終了する。

(3) リストアウト対象になるストラクチャへのポインタを抽出する。

(4) 抽出したポインタをもとに、そのストラクチャのリストアウトを行う。　この動作は、実記憶上（作業領域を使用）で行う。詳細は次項で述べる。

(5) リストアウトのストラクチャのサイズが、一定の大きさ（以後Ｓｓで表す）に満たない場合、(3) に戻る。

(6) 実記憶上にあるリストアウトされたデータを実際にディスク上に書き込む。

(7) リストアウトを終了する。

4.1.2. リストアウト時の実記憶上での作業

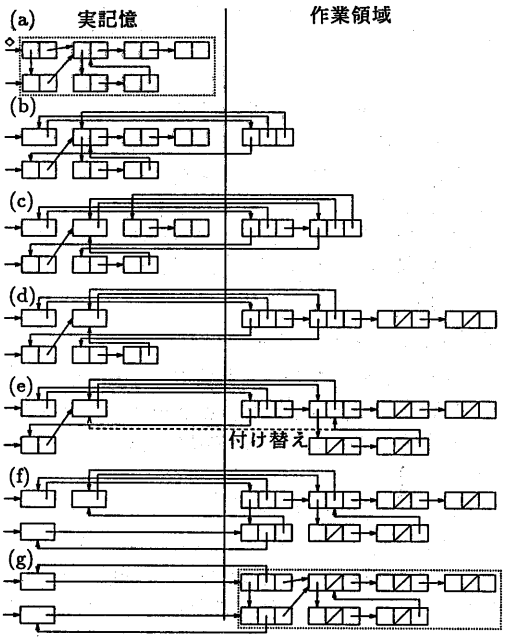


図６．リストアウト処理時の実記憶上でのノードの動き

図６は処理中のノードの動きを示している。リストアウト時の実記憶上での作業は以下の順序で行われる。

(1) 抽出されたストラクチャへのポインタをベースポインタとする。　[図６－ａ中、◇がベースポインタである]

(2) ベースポインタが指しているノードを間接ポインタを用いて作業領域に移す。そのノードが持っていたポインタを次のベースポインタとする。[図６－ｂ]

(3) ベースポインタが指しているノードの状態により動作が異なる。

・他に参照しているノードがある場合。
　間接ポインタを用いて作業領域に移す。[図６－ｃ]

・他のどこからも参照されていない場合
　そのまま作業領域に移し、ポインタを張り替える。[図６－ｄ]

(4) (3) でのノードが持っていたポインタを次のベースポインタとする。ただし、その先が間接ポインタであれば、ただちに付替る。[図６－ｅ]
さらに、その間接ポインタへの参照がなくなれば、２次記憶上からの逆ポインタを削除する。[図６－ｆ、ｇ]

(5) リファレンスカウンタを書き換える。

(6) (4)でのベースポインタをもとに (3) へ戻る。これは再帰呼び出し等で実現可能である。

(5) そのストラクチャの処理がすべて終了するか、処理サイズがＳｓに達したとき、終了する。

4.2. リスト・イン

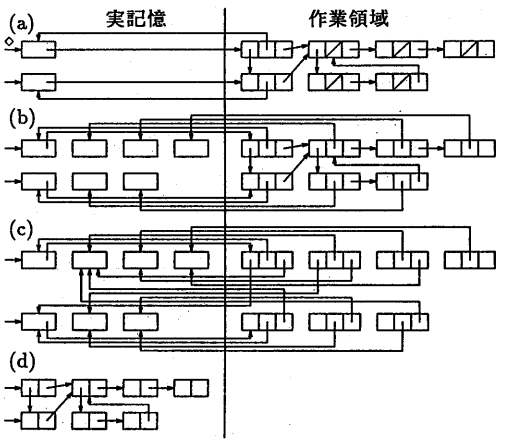


図７．リストイン処理時のノードの動き

図７は処理中のノードの動きを示している。リストイン時の作業は以下の手順にて行われる。

(1) アクセスしようとするノードが実記憶上にない場合、すなわち間接ポインタであった場合には、リスト・インが起動される。

(2) リスト・インする領域のノードを２次記憶から実記憶上の作業領域に移す。[図７－ａ]

(3) 作業領域に移したノードをリニアサーチして、それに対応するノードを、実記憶上に作る。この時点では、まだ総てのポインタを付け替えず、２次記憶上のノードの逆参照ポインタ部分のみ

を変更する。[図７－ｂ]

(4) ポインタを付け替る。ノード相互間のポインタを張り替える。しかし実記憶上のノードから外部に出る参照はない。[図７－ｃ]

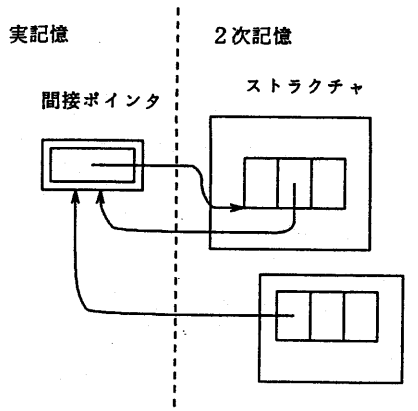
(5) ノードのデータ本体を移す。ここで、実記憶上にノードが完全に移される。[図７－ｄ]

(6) リスト・インを終了する。

4.3. ストラクチャ間のノードの参照

ＳＯＳＯの処理の際には、２次記憶上にリストアウトされたストラクチャ内のノードから実記憶上のノードへの参照がしばしば生じる。この内、図８－ａに示したように、２次記憶上から参照されている実記憶上のノードが間接ポインタであり、かつそれが実記憶から参照されていない場合、図８－ｂに示したように、この間接ポインタへの２次記憶からの参照をすべて書き換え、間接ポインタを削除することも可能である。

(a) 実記憶上に間接ポインタが残る状態



(b) ポインタを張り替え、間接ポインタを削除した状態

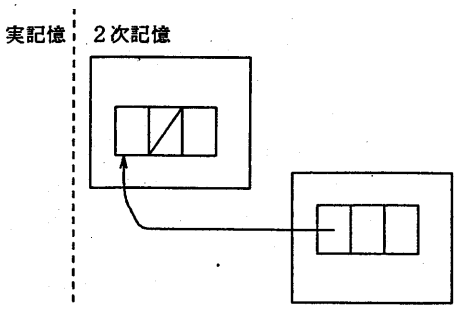


図８．リストアウトされたストラクチャ間の参照

この一連の操作は、原理的には、各々の間接ポインタへの逆参照表を作成し、リストアウトの際にリストアウトされたストラクチャ内の、該当する間接ポインタへの参照を書き換えることで実現可能である。

しかし、上記の操作は、２次記憶内のストラクチャを操作する動作を含んでおり、２次記憶へのアクセスが増加する可能性がある。そこで、リストアウトされたストラクチャ間の参照をまとめて扱い、２次記憶へのアクセスを減少させるために、以下に示す各テーブルを用いる。

[ＸＲＴ](Exterior Reference Table)

リストアウトされた１ストラクチャにつき１つのＸＲＴが割り当てられる。対応するストラクチャ内で、そのストラクチャの外部を参照しているノードとポインタが格納される。[図９－ａ]

[ＳＲＴ](Structure Reference Table)

リストアウトされたストラクチャをまとめて管理する。各ストラクチャ内部のノードが、その外部から参照されている数（リファレンスカウント）と、ＸＲＴへのポインタが格納される。[図９－ａ] このテーブルは、ＸＲＴの管理、検索などに用いる。

[ＤＲＴ](Dereference Table)

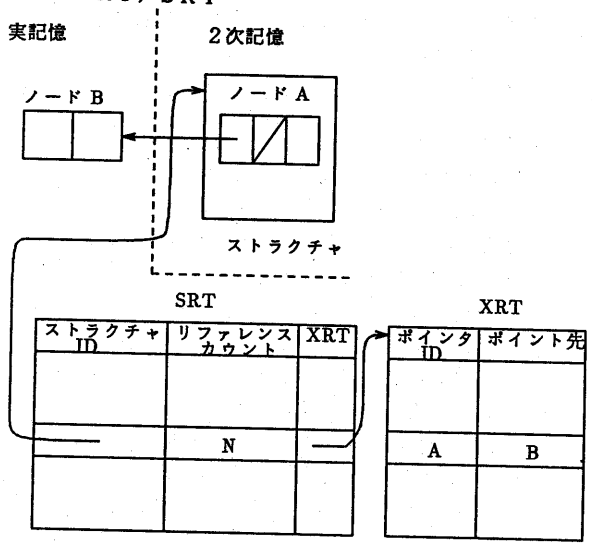
２次記憶から実記憶への参照を管理する。２次記憶から参照されている実記憶上のノードと参照しているノードが格納される。[図９－ｂ]

ＤＲＴ上に登録されているノードがリストイン／リストアウトされたとき、その対応するＤＲＴのエントリを書き換え、そのノードを参照しているストラクチャのＸＲＴの外部参照ポインタを書き換える。

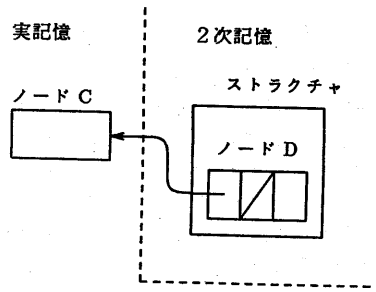
これらのテーブルを使用すれば、実記憶上に不必要な間接ポインタが残存することがなく、処理効率の向上が期待できる。

ここで、各テーブルのエントリは、ＳＲＴはストラクチャ番号、ＸＲＴはストラクチャ内アドレス、そしてＤＲＴは実記憶アドレスによりハッシュすることが可能であり、テーブルの検索などを高速化することが可能である。ただし、被参照ノードが実記憶にないとき、ＤＲＴへのハッシュは不可能であるが、ストラクチャ内のノードからの逆参照ポインタを用いて、それをＤＲＴのエントリとすることができる。

(a) XRT, SRT



(b) DRT



DRT

外部参照セル	参照元ストラクチャーノード
C	S-D

図9．XRT，SRT，DRT

5. SOSO方式の評価

5.1. 仮想記憶システムの処理効率

いま、2次記憶と実記憶間の1ノード当たりの平均転送時間を Ts 、SOSOでのポインタの張り替え等、仮想化に必要な処理に要する1ノード当たりの平均時間を Tms 、全体の転送ノード数を Ns 、ページング方式でのページテーブルの書き換えなど仮想化に必要な処理に要する1ノード当たりの平均時間を Tmp 、全体の転送ノード総数を Np とすると、仮想化のために要した時間はSOSO、ページング方式それぞれ

$$SOSO:\ Ns\ (Ts + Tms) \qquad [1]$$

$$Paging:\ Np(Ts + Tmp) \qquad [2]$$

により表すことができる。ここで一般的に、それぞれの方式の処理時間が

$$Ns\ (Ts + Tms) < Np(Ts + Tmp) \qquad [3]$$

なる関係であれば、SOSO方式による仮想記憶処理効率がページングのそれに比べて改善できる可能性があると考えることができる。

[3]式が広範な条件下で成立するか否かを知るうえで、同式の Ns , Np の値の測定は重要な示唆を与えてくれる。

本研究では、 Ns , Np の値の測定を、両方式に対して、公平かつ正確に行うため、以下のシミュレーションを行った。

5.2. シミュレーションの概要

シミュレーションは、実際の処理系に近い形で実行させるよう配慮した。そのため、シミュレーションを行うにあたり、以下に示す2点に留意した。

- 実際のアプリケーションを実行させる。
- 実際の処理系にシミュレータを組み込む。

これにより、開発したシミュレータは、ワークステーション NEWS 上で、すでに我々が作成し稼働していた Lisp 1.5 [5] インタプリタに埋め込む形で実現することとした。実行させるアプリケーションは、Lisp コンテスト [6] 等の一般的な問題とした。 Lisp 1.5 インタプリタの概要は以下の通りである。

- 総セル数：１００ｋセル
- 記述言語：C
- ポインタ幅：３２ビット
- 変数束縛方式：ディープ・バインド方式
- 組み込み関数：約１５０個

シミュレーション上では、実記憶上のノードや仮想的なページに対し、それが2次記憶上に存在するか、実記憶上に存在するかを示すフラグを設けて実現しており、実際の2次記憶装置は使用していない。リストフォールト、ページフォールトの検出は、このフラグの値により判断を行い、リストイン／リストアウト、ページイン／ページアウトをシミュレートしている。

さらに、2次記憶と実記憶間の1回のデータ転送量が、仮想記憶の処理効率に影響するため、シミュレータではこのサイズ（以下ブロックサイズという）を適宜変更して測定可能とした。

5.3. シミュレーション結果

シミュレーションの一例としてハノイの塔[7] (HANOI-9) と Lisp コンテストの問題 BITB-8 についての結果を図１０に示す。縦軸には実記憶と２次記憶間の総転送セル数、横軸には２次記憶のブロックサイズが示してある。この結果から、ＳＯＳＯ方式ではページング方式に比べて２次記憶とのデータ転送量がおおよそ２から４倍程度少ないことがわかる。

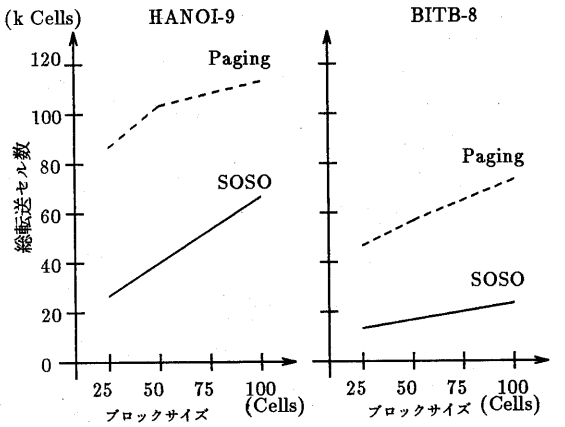


図１０．ＳＯＳＯ方式とページング方式とにおける実記憶・２次記憶間転送セル数の比較

ここで、シミュレーション結果より

$$Np = \alpha\, Ns \quad (2 \leq \alpha \leq 4) \qquad [4]$$

とする。また、

$$Tms = n\, Tmp \qquad [5]$$

とすると、[3]式より

$$n(Ts + Tms) < \alpha(n\, Ts + Tms)$$

$$Tms < \frac{n\,(\alpha - 1)}{n - \alpha}\, Ts \qquad [6]$$

が得られる。いま、 Tms が Tmp に比べて十分大きく、ＳＯＳＯ方式にとって処理効率の面から最悪の条件を考え、 $n \to \infty$ とする。これにより、[6]式の右辺の係数は、

$$\lim_{n \to \infty} \frac{n\,(\alpha - 1)}{n - \alpha} = \alpha - 1 \qquad [7]$$

となる。[4]式の α を、シミュレーション結果の中間値をとって、

$$\alpha = 3 \qquad [8]$$

と仮定すると、[6],[7]式から

$$Tms < 2\, Ts \qquad [9]$$

が得られる。現在の一般的な２次記憶装置では Ts の値として　20〜50(ms)　が得られる。ここで $Ts = 30$ (ms) と仮定すると、[9]式は

$$Tms < 60\ (ms) \qquad [10]$$

となり、これを満たせば、ＳＯＳＯ方式がページング方式に比べて、処理効率が良いと言える。

一般的に Tms の処理は 60(ms) で十分終了できると思われ、式[10]の条件は、十分満足させることが可能であると考えられる。

本シミュレーションは、ＳＯＳＯ方式では実記憶上の範囲でＧＣが適宜行われているのに対してページング方式ではＧＣを行っていない。実際のページング方式ではＧＣを行う必要があり、いかなるＧＣ手法を選択したにせよ、２次記憶と実記憶とのデータ転送量は増加する可能性が大である。以上より、式[10]の右辺に示される値はさらに大きくなり、ＳＯＳＯ方式が、よりリストデータの仮想記憶システムにとって有利と思われる。

6. まとめ

本研究では、記号処理分野で多用されるリストデータの仮想記憶管理の一手法として、その処理効率を向上させることが期待できる、ＳＯＳＯと呼ぶ方式を提案した。さらに簡単なシミュレーションによって、リスト処理においての本方式のページング方式に対する一応の優位性が認められた。今後、この方式を実際の処理系にインプリメントし、詳細な性能評価を行っていく予定である。

[謝辞]

本研究を遂行するにあたり、多大なるご指導、ご鞭撻を賜りました、大阪大学　安井先生、青山学院大学　井田先生、京都大学　柴山先生に深く感謝いたします。また、本研究の機会を与えてくださった、総合研究所　宮岡所長、吉田副所長に、お礼申し上げます。

【参考文献】

1: Harvey M. Deitel, An Introduction to Operating Systems, Addison-Wesley Publishing Company, pp.179-243 (1984).

2: 安井裕, LISPマシン, 情報処理学会誌, Vol.23, No.8, pp.757-772 (1982).

3: 服部彰, Lispマシン, 人工知能学会誌, Vol.2, No.4, pp.431-440 (1987).

4: David L. Andre, Paging in Lisp Programs, The Faculty of Graduate School of the University of Meryland (1986)

5: J.McCarthy, et al., Lisp 1.5 Programmer's Manual, MIT press, (1962)

6: 奥野博, 第３回 Lisp コンテスト及び第１回 Prolog コンテスト報告, 情報処理学会記号処理資料13-4(1985)

7: 中西正和, Lisp 入門, 近代科学社