計算機アーキテクチャ 85−10
（1990．11．22）

Linked Data Structures のための記憶管理とその動特性

實藤隆則、前川博俊
ソニー株式会社　総合研究所

ポインタでリンクされたデータ(リストデータ)は、記号処理の分野において重要で不可欠である。しかしリストデータは、ページング方式などの仮想記憶システムにおいては効率良く処理できない。我々は、リストデータの構造上の性質を活かした記憶構成方式、MOLDS(Memory Organization for Linked Data Structures)を考案した。MOLDSは階層的記憶構成において記憶管理やGCの処理を効率良く行なうことができ、また、ポインタのための記憶容量が少なくて済む。シミュレーションによる評価に基づき、MOLDS上でのリストデータの処理はページング方式に対し10〜40倍高速であり、また、記憶の使用効率も優れていることを確認した。

The Memory Organization for Linked Data Structures and Its Dynamic Characteristics

Takanori SANETO and Hirotoshi MAEGAWA

Corporate Research Laboratories
Sony Corporation
6-7-35 Kitashinagawa
Shinagawa, Tokyo 141
Japan

Linked data structures are important and indispensable in the area of symbolic computation. However, manipulating such sturctures is not suitable for virtual memory such as a demand-paged system. We propose a memory organization scheme called MOLDS (Memory Organization for Linked Data Structures), which is based on the structural characteristics of linked data structures. MOLDS is capable of efficiently processing memory management and reclamation on a hierarchy of storage devices and effectively decreasing memory consumption. We confirmed by software simulation that list processing in MOLDS was between 10 and 40 times faster and that MOLDS was more efficient for the memory space consumption.

1. はじめに

記号処理の分野では、ポインタによってリンクされたデータ構造 (以下リストデータと呼ぶ) を用いることが多い。リストデータを用いることで複雑なデータ構造を構築することやその構造の変更などが柔軟かつ効率よく行なえるためである。

その柔軟性から、リストデータのもう一つの特徴としてデータを計算機のデータ空間上にマップしたさいに、アクセスされるアドレスに局所性がほとんどないことがあげられる。これは、数値計算等で多用されるベクトルデータとは大きく異なった性質である。

近年記号処理の世界では、汎用機上に記号処理システムをインプリメントする場合はもちろんのこと、専用機の場合でも大規模な問題を扱う必要性から、階層型の構成による記憶システムの仮想化が当然のように行なわれるようになってきている。この場合、仮想記憶の方式としては、ページング方式が使われることが多い。

ページング方式はアクセスされるデータのアドレスに局所性があることを前提とした方式である。しかし上記のように、リストデータの処理においてはこの前提は成り立たないことが多く、そのため不必要なデータが主記憶と二次記憶との間でやりとりされることになる。また、ガーベジコレクションの処理では一般に全記憶空間のアクセスが必要であることから、大量の二次記憶アクセスが発生し、これもまたシステムの性能を低下させる一因となっている。

そこで我々は記憶空間の構成を根本から見直し、リストデータの特性をそのまま活かした記憶構成方式MOLDS(Memory Organization for Linked Data Structures)を考案した[1][2][3][4]。MOLDS では相互に関連の深いリストデータをまとめ、これを主記憶と二次記憶との間の転送単位とする。このため、二次記憶に書き込まれるデータにガーベジが含まれることはない*。また、ガーベジコレクション(GC)も二次記憶上におかれたデータ自体にはアクセスすることなく実現できるため、これによって発生するデータ転送もない。さらにポインタは、仮想記憶空間全体としてではなく主記憶と二次記憶とのそれぞれの空間で表現するため、二次記憶上でのデータ量が小さく、また、両者のアドレス空間を独立に拡張可能であるという特長を持つ。

今回我々はMOLDSの記憶管理アルゴリズムを検討するためのシミュレータを作成した。同時に、従来のページング方式と比較するため、ページング方式を用いるシミュレータも作成し、実際に比較を行なった。その結果、 MOLDSの有効性を示すデータが得られたので報告する。

* 読み出し時に転送するデータにはガーベジが含まれることもある。

2. MOLDS の概要

階層型の記憶構成による記憶システムの仮想化の手法として従来広く使われてきた方法にページング方式[5]がある。ここでは、記号処理の分野にこの手法を適用した場合に生じる問題点について述べ、さらに、我々が提案する記憶システムの構成方式MOLDSについて述べる。

2.1. 記号処理システムとページング方式

ページング方式は、従来汎用計算機上で仮想記憶システムを実現する手法として広く用いられており、十分な実績を持った方式であると認識されている。

この方式では、「プログラムの論理的なデータ空間においてアドレスの近接したデータは論理的なつながりが強く、したがってあるアドレスのデータがアクセスされた場合、その近くのデータがアクセスされる可能性が高い」という仮定の下に構成されている。実際、数値計算などの応用分野では配列などのベクトルデータを用いることが多く、上記の仮定が満たされているために、ページング方式が有効に機能する。

これに対し記号処理の分野では一般にはポインタによってリンクされた形式のデータ(リストデータ)を用いることが多い。リストデータでは、データの間の論理的つながりはポインタでリンクされていることのみであり、そのデータが置かれているデータ空間上の位置には論理的な意味付けは全くない。さらに、プログラムの実行にともない、データ構造が柔軟に変更されていくため、結果として、プログラムからアクセスされるデータのアドレス系列はランダムなものとなりやすい。

ページング方式においては二次記憶とのデータの転送をアドレスに基づいたページという単位で行なっているため、データがランダムにアクセスされると、記憶の階層間を転送されるデータに無駄が多くなりシステムの性能の低下を招く。また、リストデータを扱う場合、それ以降参照されることのないデータ、いわゆるガーベジも他の有用なデータと同様に記憶階層間を転送されるため、更に効率が悪くなる。最悪の場合、主記憶上にないデータばかりがアクセスされたとすると、データのアクセス速度は二次記憶のアクセス速度まで落ちてしまうことになる。

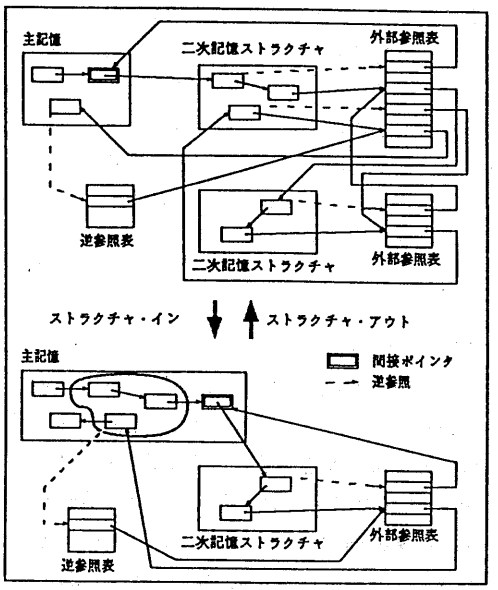
データのアクセスがランダムであるということの他にも、GC という問題がある。GC は記憶領域の管理を自動化し、ユーザの手間を大きく減らすものであるが、その処理のためには、一般に全記憶領域のアクセスが必要となる。このためページング方式では、GCのたびに記憶階層間で大量のデータがやりとりされる。このことによる全体の処理効率の低下を避けるため、コピー方式やコンパクション方式のGCを用いるなど種々の工夫がなされている[6][7][8]が、いずれも本質的な解決策とはなっていない。

2.2. 記号処理向き記憶管理方式 MOLDS

以上のように、一般の仮想記憶システムで有効と考えられているページング方式であるが、リストデータを多用する記号処理の分野においては不都合な点が多いことがわかる。そこで我々は記憶の構成方式を根本から見直し、リストデータの性質をそのまま活かした記憶管理方式を考案し、これを MOLDS (Memory Organization for Linked Data Structures) と名付けた。

MOLDSでは、記憶階層間のデータの転送単位として、互いに論理的つながりの深いデータの集まり(以下これをストラクチャと呼ぶ)を用いる。主記憶上のリストデータからストラクチャを抽出し、二次記憶に掃き出すことをストラクチャ・アウトと呼び、逆にストラクチャを二次記憶から主記憶に転送することをストラクチャ・インと呼ぶ (図1)。これらは、ページング方式におけるページ・アウト、ページ・インに対応する概念である。

ストラクチャを構成するデータは互いに論理的つながりが深いため、一旦あるストラクチャが主記憶上に読み込まれると、そのストラクチャを構成するセルがその後連続してアクセスされることが期待できる。またストラクチャ・アウトされるデータはガーベジを含まないため、ページング方式で問題となった無駄なデータの転送はMOLDSでは起こらない†。



外部参照表はさらにストラクチャ参照表（図では省略）で管理する。

図1. ストラクチャ・アウト/ストラクチャ・イン

ストラクチャ間、あるいは主記憶からストラクチャへの参照、被参照は、ストラクチャとは別に管理する(図1の参照表)ことで、ストラクチャ・イン、ストラクチャ・アウトの際の二次記憶アクセスを最小限にすることができる。　またGCについても、二次記憶上のデータに関してはストラクチャ単位でGCをおこなうことで、主記憶上のみで行なうことができる。

MOLDSでは主記憶と二次記憶上のストラクチャとでは異なったデータ空間となっているため、ストラクチャの抽出にともなって主記憶空間からストラクチャへのポインタの変換を行なう。このとき、ストラクチャ内のポインタはストラクチャの大きさの空間を指せるだけの幅を持つだけでよいため、結果として、二次記憶装置上でセルあたりに必要となるデータ量はページングに比べかなり小さくなる。

† 複数のストラクチャをまとめて転送単位とした場合には、あるデータをアクセスするためにそれと論理的につながりのないデータが一緒にストラクチャ・インされてくる場合が生じる。

以上のような点から、MOLDSではページング方式に比べ、記憶階層間のデータ転送量を相当量小さくすることが可能である。二次記憶のアクセスは主記憶のアクセスと比較すると非常にコストの高いものであるから、ストラクチャ抽出やポインタ形式の変換にともなうオーバヘッドを考慮しても全体としてMOLDSの方がページング方式よりも処理速度が向上することが期待できる。

3. シミュレータ

MOLDSにおけるストラクチャ抽出のアルゴリズムの評価およびページング方式との比較のため、簡単な記号処理システムを作成し、その記憶管理部をMOLDSおよびページング方式を用いて実装した。シミュレータの構成を図2に示す。以下では、今回作成したシミュレータをSimu/M(MOLDS)およびSimu/P(ページング方式)と呼ぶことにする。

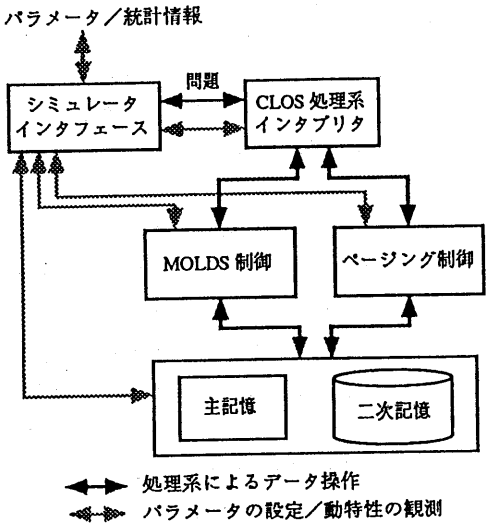


図2. シミュレータの構成

シミュレータの言語処理系部分には、CLOS[9] サブセットの処理系（インタプリタ）を実装した。データタイプとしてはシンボル、整数、文字、コンスおよび文字列をサポートしている。シンボルやオブジェクトは特殊なリストとして実現されており、文字列はコンスセルと同じ領域に埋め込まれているので、処理系の記憶空間は実質的にコンスセルのみからなる。

記憶管理部はMOLDSあるいはページング方式の処理を行なう。実行にあたっての各種のパラメータは起動時にファイルによって与える。以下にそれぞれの記憶管理部について説明する。

3.1. MOLDS

MOLDSにおけるストラクチャ・アウトはGCを行なったあと主記憶上に十分な量のフリー・セルが得られなかった場合に起動する。シミュレータでは、ストラクチャ・アウトが起動されるセルの量やどれだけのセルが解放されるまでストラクチャ・アウトを続けるかなどをパラメータとして与えることができる。シミュレーションはいくつかのパラメータについて行なったが、結果に大きな差は見られなかった。そのため、データとしては GC 後のフリー・セルが全体の 1/8 未満になった時点でストラクチャ・アウトを起動し、1/4 の領域を解放するという設定でのもののみ示した。MOLDS では、一括型の GC に加え、リアルタイム GC を併用している。通常リアルタイム GC では一括型のGCに比べて記憶管理のオーバヘッドが大きく、処理速度は遅くなりがちである。しかし MOLDS の場合ストラクチャ・アウトの処理においてセルへの参照の有無を管理する必要があるため、これをリアルタイム GC の処理と兼ねることで少ないオーバヘッドでリアルタイム GC が実現できる。

ストラクチャ抽出のアルゴリズムとしては、処理系に依存せず、ストラクチャアウトが必要とされた時点でのデータ空間状態からストラクチャを抽出するやり方が考えられる。このような方法の利点は、処理系との依存度が低いために、既存の処理系への組み込みが容易となること、また、アルゴリズムの変更が処理系に与える影響を最小限に押えることができることである。その反面、静的なデータ空間の状態からその時点での必要度がもっとも低いデータを選び出すことは容易ではなく、またコストの高いものとなることが予想される。そこで我々はある程度処理系に依存した方法でストラクチャ抽出を行なうことにした。

Simu/Mで採用したストラクチャ抽出のアルゴリズムは次のようなものである。プログラムの実行中に生成される論理的に意味のあるデータのまとまり(関数の評価環境、制御環境、シンボルの実体など)に注目し、これらをストラクチャの候補とする。これらの候補を各時点での必要度を反映しさせて順序付けしたリ

スト(候補リストと呼ぶ)として保持し、ストラクチャ抽出の際には候補リスト中の順位の低いものから抽出の対象とする。順位の付け方としては、環境については最近生成されたもののほど高く、またシンボルについてはLRU方式で管理する。このような順序づけにより、現在の実行環境から(論理的に)遠い環境やほとんど実行されることのない関数定義などがストラクチャとして抽出される。

抽出の候補が決まると、そこから実際に一定の大きさのまとまりをとり出す必要がある。リストデータの特性から、データ構造をあまり深くたどり過ぎると他のデータと相互に絡まりあってしまうことが考えられるため、幅優先にセルを選んでストラクチャとする。ある候補から得られるセルの数がストラクチャを構成するのに足りなければ次の候補からも同様にしてセルをとり出す。

Simu/Mではアルゴリズムの検討および評価のため、ストラクチャの抽出に用いるデータを処理系とは独立に管理する方式を取っている。このため記憶管理部により冗長な主記憶アクセスが行なわれる。実際のシステムへの実装では、このようなデータをあらかじめ処理系に含ませることで、オーバヘッドの少ない処理が可能となる。

3.2. ページング方式

ページング方式におけるページ置き換えアルゴリズムとして、Simu/P では完全 LRU 法を採用した。実際のシステムでは、ページの参照履歴の管理の手間を省くため、近似的な LRU を行なっているものが多いが、シミュレータ内では完全 LRU 法をオーバヘッド無しで行なえるものと仮定している。

GCについては半空間を用いたコピー方式とした。これは GC 後のセルをできるだけまとまった場所におくことでアクセスの局所性を向上させるためである。

3.3. 実行時間と記憶装置のモデル

シミュレータにおける実行時間は主記憶のアクセス時間を 1単位とした。さらに、十分高速なCPUをもつ計算機システムを想定し、CPU内での処理時間は記憶装置へのアクセス時間に対して十分に小さいため、実行時間に対する影響は無視できるものとした。

主記憶はコンスセルからなるものとする。一つのコンスセルは8ビットのタグを含んだ二つのポインタを持ち、1単位時間で読み書きできるものとする。
MOLDS では、二次記憶との転送単位はストラクチャである。ストラクチャ内部では、各ポインタの幅は、ストラクチャ内の他のセルをさせるだけの大きさがあればよい。これに加えて、ポインタ毎にそのポインタがストラクチャの内部を指すものか，ストラクチャ外への参照表を指すものかをあわらす1ビットとセルに対して、そのセルがストラクチャ外から指されているかどうかのマークが1ビット必要となる。したがってストラクチャを 2^k 個のセルから構成することにすると、ストラクチャの大きさは、

$$(2(k+1+8)+1)2^k \ \ bits$$

となる。

ページング方式の場合、主記憶上のセルの大きさがページ即ち二次記憶装置との転送単位の大きさに影響する。そこで、セルに含まれるポインタは仮想記憶空間をアドレッシングできる最小の大きさを持つものとした。1ページ(2^pセルとする)の大きさは、仮想記憶空間の大きさを 2^v とすると、

$$2(v+8)2^p \ \ bits$$

となる。表1に MOLDS でのストラクチャおよびページング方式(仮想記憶空間の大きさを５１２Kセルとする)でのページの具体的な大きさを示す。

表1. 転送単位のサイズ(単位 bit)

転送セル数	MOLDS	ページング
64	1984	3456
128	4224	6912
256	8960	13824
512	18944	27648

二次記憶としては、一般的なディスク装置を想定した。現在の SCSI ディスクでの平均シーク時間、回転待ち時間およびディスクの全容量に対する仮想記憶のために使われる領域の比率から、平均的なアクセス速度を 10ms 程度と見積もった‡。これに対し、CPUの

‡ シーク時間は使用する領域の大きさに比例して変化するが、回転待ち時間は変化しない。

半導体メモリへのアクセス速度は 100ns 程度であるので、シミュレータではディスクアクセスには 10^5 単位時間かかるものとしている。これにさらにデータの転送時間を加えたものが二次記憶アクセスに要する時間となる。データ転送時間としては転送されるデータ 1ビットあたり 0.5単位時間とした*。

なお、二次記憶装置のアクセスにおけるバッファリングやヘッドスケジューリングなどの高速化の手法については今回は考慮しなかった。

4. シミュレーション結果

4.1. パラメータの説明

前節で述べたようなシミュレータを用いて、実際に処理系上でプログラムを動かし、データの採取を行なった。図3にシミュレーションで用いたパラメータの説明を示す。

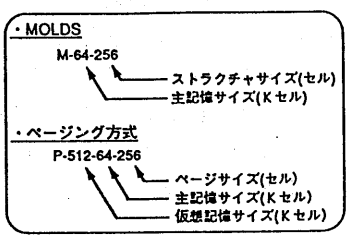


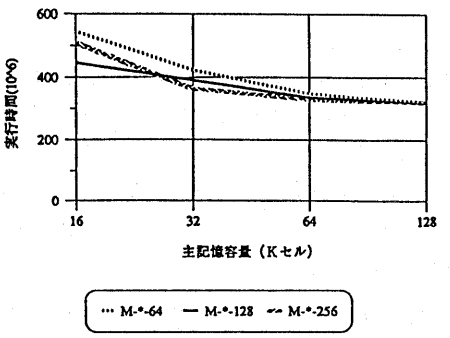
図3. シミュレーションのパラメータ

データ採取には Boyer ベンチマークを用いた。このプログラムは、CommonLisp のベンチマークの一つとして比較的よく使われているものであり、コンスセルを大量に消費する傾向がある。

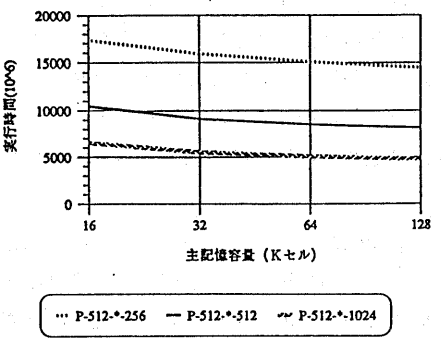
4.2. アクセス回数および実行時間の比較

図4に、パラメータをいくつか変化させた場合の MOLDS とページング方式との実行時間を、また、図5、図6に主記憶・二次記憶それぞれへのアクセス回数を示す。

これらから、以下のようなことがわかる。まず、主記憶へのアクセス回数については MOLDS がかなり多い。記憶管理部による主記憶へのアクセス回数を MOLDS とページング方式とで比較すると、MOLDS ではページング方式の7倍程度となっている。これは、

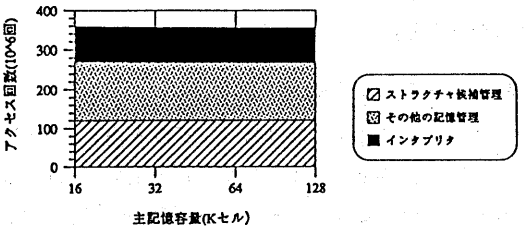


(a) MOLDS

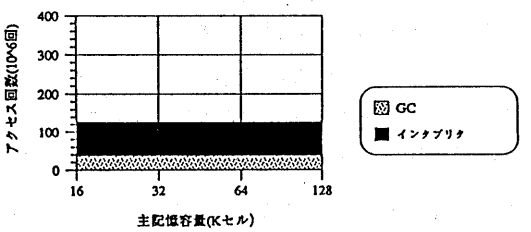


(b) ページング方式

図4. 実行時間



(a) MOLDS



(b) ページング方式

図5. 主記憶アクセス回数

ストラクチャの抽出処理のために主記憶を頻繁にアクセスするためである。ただし、このうちの半分はストラクチャ抽出のための候補リストリストの管理のため

* この数値を、主記憶のアクセス時間を100nsと仮定して転送速度に換算すると2.5Mbytes/secとなる。

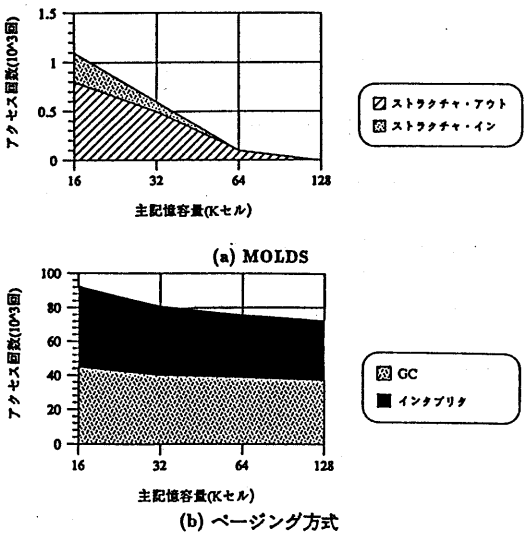


図6．二次記憶アクセス回数

のものであり、前章で述べたようにこの部分のオーバヘッドは改善が可能である。

これに対して二次記憶へのアクセス回数はMOLDSが圧倒的に少ない。二次記憶へのアクセスを回数ではなく転送データ量で比較すると、その差はさらに大きくなる。結果として、図4に示したように、MOLDSはページング方式に対して同一主記憶容量の構成で10～40倍の処理速度が得られることがわかる。

4.3. 二次記憶使用量の比較

MOLDSにおける二次記憶の最大使用量をビット数に換算したものを図7に示す。この図から、MOLDSでは主記憶に収まらないデータ量に応じて必要とされるだけの二次記憶しか消費しないことがわかる。またその使用量もページング方式に比べ実質的に半分以下であり、MOLDSの二次記憶の使用効率が非常によいことがわかる。

4.4. ストラクチャサイズの影響

図4(a)においてストラクチャサイズと実行時間の関係を見てみると、ストラクチャサイズは実行時間にほとんど影響を与えないことがわかる。実際には、ストラクチャサイズを大きくすることにより、二次記憶とのデータ転送の回数を減らすことができ、これによって二次記憶へのアクセス時間を減らすことができるのであるが、 MOLDS ではそもそも二次記憶へのアクセス回数が少ないため、実行時間に与える影響があまり大きくない。実際、図8に示すように、実行時間に占める二次記憶アクセス時間の割合は、 MOLDS では1割～2割程度となっている。特に主記憶を128Kセルとした場合には、プログラムの消費するセルすべてを主記憶でまかなうことができるため、二次記憶を一切アクセスする必要がない。これに対してページング方式では二次記憶アクセス時間が実行時間の大半を占めている。

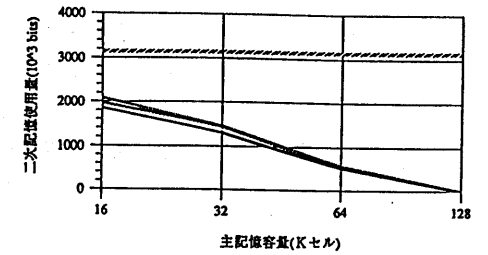


図7．二次記憶の使用量

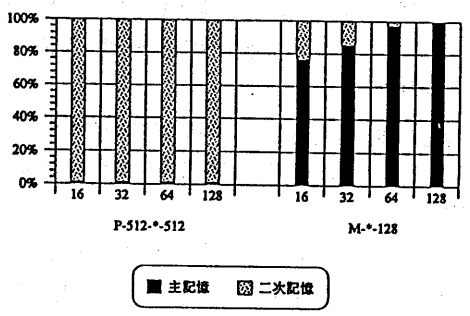


図8．実行時間のうちわけ

5. おわりに

我々の提案する記号処理向き記憶管理方式 MOLDS について、シミュレーションによってその有効性を示した。結果として、 MOLDS を用いることで、ページング方式に比べ、10～40倍の処理速度の向上が見込まれることが確認できた。

現在採用しているストラクチャ抽出のための方式はまだ試験段階であり、主記憶アクセスに関してかなりのオーバヘッドがあるが、その状態でも MOLDS はページング方式に対して相当の優位性を示している。記憶管理部のアルゴリズムを改良することにより、より一層の効率向上が期待できる。

今回のシミュレーションで設定した記憶システムの規模はそれほど大きくないが、シミュレーション結果には両方式の特徴がよく現れており、実際的な大きさのシステムでも同様な傾向が生じると考えられる。

MOLDSは既存の多くのシステムにおいても記憶制御部を変更することにより組み込みが可能である。また、記憶領域を分割することなどによって、問題に応じたページング方式との使いわけや併用を行なうことも可能である。

今後、より詳細なシミュレーションによってストラクチャ抽出アルゴリズムやデータ構造の検討を行なっていく予定である。さらに、シミュレーションで得られたデータを元に、我々が開発を進めている記号処理計算機 Lilac[10] 上に実際に MOLDS を用いた記号処理システムを実装し、実機上での有用性を検証していく。

謝辞

本研究の遂行にあたっては、福田譲治部長をはじめとする当研究部のメンバーの支援を受けた。特にシミュレーションデータの採取に関する安田弘幸氏との討論は大変有益であった。

本研究の機会を与えて頂いた当社総合研究所の宮岡所長、ならびに総合研究所情報通信研究所の松田所長に感謝する。

シミュレータの作成およびデータの採取にあたっては、(有)アクセスの村田典幸氏に御協力を頂いた。

[参考文献]

[1] 前川博俊 他: Pointer-Linked Data における仮想記憶管理の一手法, 情報処理学会研究会資料, SYM50-1 (1989).

[2] 前川博俊 他: Linked Data Structures の記憶管理, 情報処理学会第39回全国大会, 1Q-4, pp.1279-1280 (1989).

[3] 前川博俊 他: Linked-Data のその構造に基づく記憶空間の構成, 情報処理学会研究会資料, ARC80-5 (1990).

[4] Maegawa, Hirotoshi: Memory Organization and Management for Linked Data Structurs, to appear in ACM 1991 Computer Science Conference (1991).

[5] Denning, P. J.: Virtual Memory, Computing Surveys, Vol.2, No.3, pp.153-189(1970).

[6] Cohen, J.: Garbage Collection of Linked Data Structures, Computing Surveys, Vol.13, No.3, pp.341-367(1981).

[7] 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).

[8] Ungar, D.: Generation Scavenging: A Non-desruptive High Performace Storage Reclamation Algorithm, Proceedings of Software Engineering Symposium on Practical Software Developement Environments, pp.157-167(1984).

[9] Bobrow, D., et al.: Common Lisp Object System Specification, X3J13 88-002R(1988).

[10] 安田弘幸 他: 計算資源指向型並列分散処理システム — Lilac —, 情報処理学会研究会資料, SYM55-2 (1990).