

LISPでの並列処理における動的特性と EVLIS マシンの構成

安井 裕, 斎藤年史, 三石彰純, 宮崎洋一
(大阪大学 工学部)

1. はじめに

現在, 製作されているマルチプロセッサによるLISPシステム^{[1]~[7]}はリスト処理とG.C., あるいはI/O処理等を並列的に処理するものであって, リスト処理そのものは, 並列化していない。我々は, LISPの高速化を計るために特にリスト処理の並列処理化を検討し, evlisの引数評価を複数台のプロセッサで行なうLISPマシンシステムを設計した。本報告書では, 我々の並列処理の概要と, EVLISマシンと名付けた試作中の処理システム, 及び本方式の並列処理における処理速度向上のためのパラメータに対するシミュレーションの結果について述べる。

2. 並列処理の概要とデータ構造

2.1 並列処理の対象と特徴

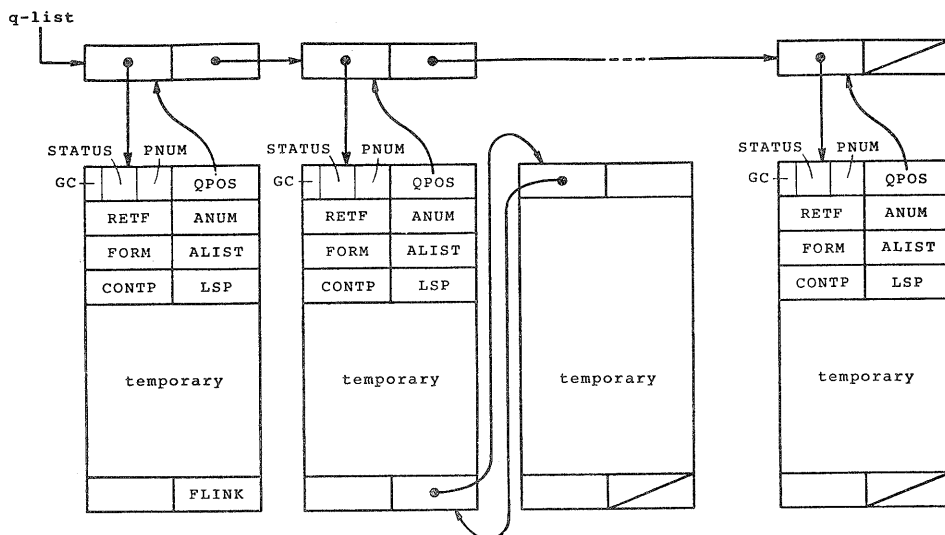
LISPインタプリタを並列化する場合, 並列化の対象と方法は幾つか考えられる。我々は, 関数 evalis のもつ次の様な特徴に着目して, その引数評価を複数台のプロセッサで行なうことにした。(1)仕事の大部分を第一引数の要素の数に分割する事ができ, それがかなりまとまった仕事(eval)である。(2)cond, and, or等と違ってその仕事の結果が全て必要である。(3)頻繁に使用される。

このシステムでは, 複数台のプロセッサは互いに対等とし, 仕事の割り当て等のOSの仕事をも各プロセッサが並列的に行なう。G.C.はリスト処理と並列には行なわれないが, G.C.が必要となった時は全プロセッサが仕事を一時中断し, G.C.の処理を並列的に行ない, G.C.終了後, 一斉にもとの仕事を再開継続する。

2.2 環境の表現とその管理

まず, これから用いる用語の約束をしておく。1回のevalの処理, すなわち eval が呼び出されてからその eval の値が得られるまでの過程をプロセスと呼ぶ。eval の処理中にあらたに eval を呼び出した場合, 呼び出した方を親プロセス, 呼び出された方を子プロセスと呼ぶ。evalis では同時に幾つかのプロセスを生成する。この場合, そのプロセス相互は兄弟プロセスであり, 第一要素のプロセスを長男, 第二要素を次男, ...と呼ぶことにする。各プロセス間には要素の順序にしたがった優先順位があるものとする。即ち, 次男より長男の方が優先順位が高い。

インタプリタの実行中プロセスは多数存在し, 各々固有の環境を保持しなくてはならない。そのため, 変数の束縛には a-list を用い, 作業用スタックは図1の様な固定サイズのスタックフレームを用いる。フレームがあふれた時はリンクして拡張する。始めのフレームを基本フレーム, 拡張したフレームを拡張フレームと呼ぶ。フレームはインタプリタ起動時及び evalis によるプロセス生成が行なわれた時に作成し, 基本フレームの固定領域にはそのプロセス実行に必要な情報を格納する。g-list はその時点で存在する全てのフレームを, 対応するプロセスの優先順位の順に従ってつないだもので, プロセスの処理状況を把握するために用いる。新しいフレームを作成する時は優先順位に従って g-list 上に



STATUS プロセスの状態を示す
PNUM このフレームを使用しているプロセッサの番号
QPOS `q-list`へのポインタ
RETF 親フレームへのポインタ
ANUM 兄弟プロセスの順位
FORM, ALIST このフレームで評価される `eval` の引数
CONTP このフレームでの処理再開番地
LSP このフレームのローカルスタックポインタ
FLINK 拡張フレームへのポインタ

図1. 環境の表現と `q-list`

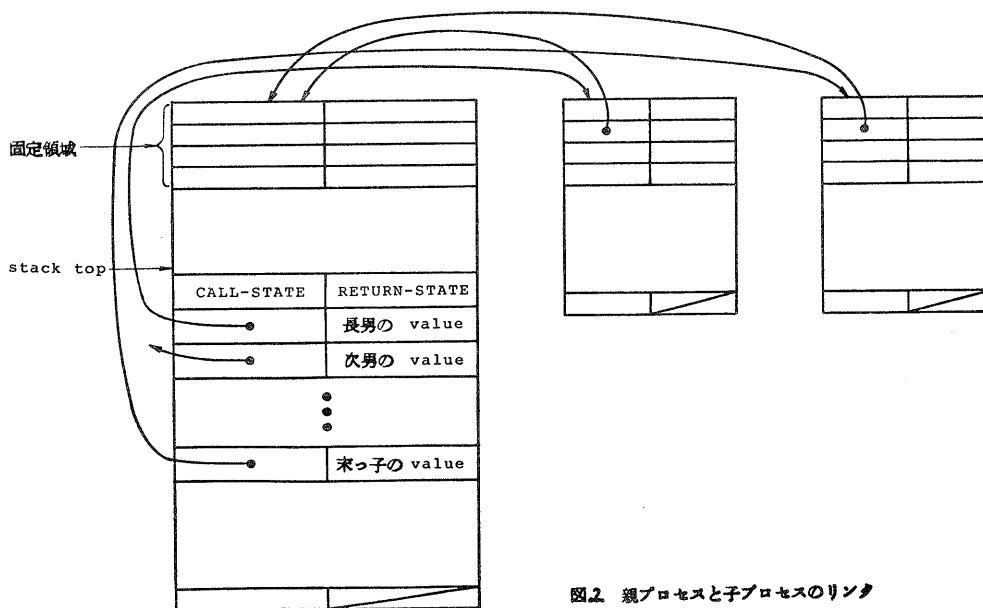


図2. 親プロセスと子プロセスのリンク

登録する。フレームはその作成者である親プロセスが不要と認めた時消滅する。

2.3 インタプリタの動作

図1においてSTATUSにはそのフレームを使用しているプロセスが、1.処理待ち、2.実行中、3.子プロセスの値待ち、4.完了のいずれの状態にあるかを示すフラッグを格納する。システムを初期化した時、*g-list*はNILにする。関数 *read* によりダブレットが入力されると、*evalquote* はそれをもとにフレームを作成し、*g-list* に登録する。この時、このプロセスがトップレベルである事示すために、フレーム内の RETF を 0 にする。また *g-list* に登録する時は STATUS = "処理待ち" とする。アイドル状態のプロセッサは *g-list* を最初から調べて、STATUS = "処理待ち" のフレームを探す。見い出したならば、STATUS に "実行中"、PNUM に自分のプロセッサ番号を書き込んでプロセスの処理に入る。"処理待ち" のフレームを見い出せなかった場合は、再び *g-list* を最初から探す。なお、STATUS のロケーションは *test and set* 命令の様に排他的に調べる必要がある。RETF = 0 のフレームの処理を終えたプロセッサは、値を関数 *print* に渡して結果を出力する。

2.4 *evalis* の処理

本システムにおける *evalis* は、並列処理の中心をなすもので次の様な機能を新たに追加変更したものである。第一引数の要素がない場合は NIL を値として帰り、要素が1個以上(2.6で述べる *strategy* 3,4 の場合は2個以上)ある場合には、各要素の評価に必要な情報を格納したフレームを作成し、*g-list* 上に登録する。親プロセスのフレーム内のスタックには、図2の様に CALL-STATE、RETURN-STATE、子プロセスへのポインタ、及び子プロセスの値を入れるフィールドを用意する。CALL-STATE には子プロセスの数を格納する。RETURN-STATE の各ビットは最下位より長男、次男、...に対応し、生成した子プロセスのビットを1にしておく。これらの準備をした後、親プロセスの処理は子プロセスの値が全て求まるまで中断する。子プロセスが終了した時は一旦親プロセスへ戻り、求まった値を親プロセスのスタック上にある子プロセスの値を入れるフィールドに格納し、同じく RETURN-STATE の子プロセスに対応するビットを0にする。その結果 RETURN-STATE の全ビットが0であれば、子プロセスは全て完了しているので親プロセスの処理を再開する。RETURN-STATE が0でなければ、親プロセスは再開できないので、そのプロセッサはアイドル状態となる。なお、RETURN-STATE は STATUS 同様排他的に操作する必要がある。

2.5 リスト構造を書き換える関数

setq, *rplaca* 等のリスト構造を書き換える関数は、各プロセスを並列に処理する場合、他のプロセスの結果を誤らせる可能性がある。それらの関数を含むプロセスより優先順位の高いプロセスは、その関数の実行以前の環境で処理すべきであり、また優先順位の低いプロセスは、その関数の実行後の環境で処理しなければならない。これらの関数によって必ずしも他のプロセスが誤った結果を得るとは限らないが、実行時に影響の有無を確認する事は困難であると思われる。そこで我々は、実行時にリスト構造を書き換える関数に遭遇した場合、*g-list* を一旦自分の所で切断し、その後ろ、即ち自分より優先順位の低いプロセスの中で実行中のものがあれば、その実行を中止させ、それらの結果は放棄する。その後 *g-list* を調べて自分より優先順位の高いプロセスで "実行中"、もしくは、"処理待ち" のものがなくなるまで待つて、関数の目的であるリスト構造を書き換え

る。その後、切り離しておいた *q-list* の後ろの部分を再び接続する。この時には後ろの部分のフレームの *STATUS* は全て "処理待ち" に変更しておく。これはそれらのプロセスを最初の状態から再実行させることを意味する。また、それらの中でその親プロセスがリスト構造を書き換えたプロセスより優先順位の低いものは、処理が進むに従い再び生成されるので *q-list* から削除しておく。この方法は、リスト構造を書き換える関数の実行にかなりのオーバーヘッドを要する可能性がある。しかし、実際のプログラムでは、リスト構造書き換えの大部分はローカルな変数の値の書き換えである場合が多く、ローカル変数の値の変更は他のプロセスに影響を与えないので、上記の手続きを踏まずにリスト構造を書き換えてしまう **setq*, **replace* 等の新しい関数を用意した。これをユーザーが積極的に意識して記述するか、プリプロセッサ、コンパイラ等でそれらに置き換える事を考えている。これらのプログラミング上の工夫とインタプリタの動的特性の実測値^[8]からオーバーヘッドは十分許容できるものと考えている。

2.6 プロセスへのプロセッサの割り当て

プロセスへのプロセッサ割り当てを行なうタイミングは、*evalis*によってプロセスが生成された時と、そのプロセスが終了した時が考えられる。プロセス生成を動機とする割り当てには次の4つの *strategy* が考えられる。ここで未処理プロセスとは、今まで処理待ちで残っていたプロセスと今生成したプロセスの全体を意味する。

- (1) 今、生成されたプロセスよりも、優先順位の低いプロセスが処理中であれば、その処理を即刻中断させてそのプロセッサを今生成したプロセスに割り当てる。
- (2) 親プロセスに割り当てられていたプロセッサを、未処理プロセス中最も優先順位の高いプロセスに割り当てる。
- (3) 生成したプロセスが1個の場合、親プロセスに割り当てられていたプロセッサは生成したプロセスを実行する。2個以上生成した場合は(2)の方法を用いる。
- (4) 親プロセスに割り当てられていたプロセッサを、今生成したプロセスの中で最も優先順位の高いプロセス(長男)に割り当てる。

これらの *strategy* の選択は、システムの処理効率を左右する重要な要素と考えられる。一方、プロセス終了時には2.3で述べた様に、兄弟プロセスが全て終了している場合は親プロセスを実行し、そうでない場合はアイドル状態となる。

2.7 G.C.に関して

G.C.が呼ばれると全プロセッサの処理を中断し、特定のプロセッサをG.C.マスターとして、その指示のもとにG.C.の作業を並列処理で行なう。まず、各プロセッサは自分の使用している基本フレーム、拡張フレームにマークを付け、フレーム内のスタックを根としてリストにマーク付けを行なう。アイドル状態のプロセッサ、及び自分のフレームのマーク付けを終えたプロセッサは、G.C.マスターにさらに根を要求する。

根要求を受けたG.C.マスターは、*q-list* から *STATUS* = "実行中" のフレームをとり出して要求を出したプロセッサに与える。与えるべきフレームがなくなったら、マーク付けは終了する。回収フェーズでは、各プロセッサの分担領域が定めてあるので各々自分の領域の回収を行ない、終了次第G.C.マスターに回収した自由リストと自由フレームを渡す。全プロセッサが回収を終れば、G.C.は完了し、もとの処理を再開する。

3. EVLIS マシンの概要

3.1 システム構成

試作中の EVLIS マシンは、EVAL プロセッサ群、I/O プロセッサ、メイン メモリー、入出力デバイス等から成り、その構成を図3に示す。最終的には、仮想記憶も備える事を考えているが、現段階においては実現していない。

本システムの中心は EVAL プロセッサ群で *g-list* よりフレームを得てプロセスの処理を行なう。EVAL プロセッサは最大7台接続可能である。I/O プロセッサは、EVAL プロセッサのマイクロプログラムのロード、入出力デバイスの管理の他、LISP の入出力関数、G.C. マスターを受けもつ。EVLIS マシン内のデータ長は40ビットで、リスト構造の1セルを構成する。

3.2 EVAL プロセッサ

EVAL プロセッサは、今回の試作機では、インテル社のビットスライスマイクロプロセッサ $\text{\textcircled{I}}3000$ シリーズの演算素子 3002 (2ビットスライス) 10個 (CPA)、マイクロプログラム制御素子 3001 1個 (MCU) を中心に、マイクロ命令を格納するコントロールメモリー (WCS)、高速スクラッチパッドメモリー (S-MEM)、CAR-CDR レジスタ、パイプラインレジスタ (PLR)、内部バス制御回路 (BCU)、割り込み制御回路 (ICU) 等から成る (図4)。EVAL プロセッサのデータ語長は20ビット、アドレス空間は16ビットである。

● 内部バス構成 EVAL プロセッサの内部バスは A-BUS (アドレスバス、16ビット)、M-BUS (データバス、20ビット)、K-BUS (定数バス、20ビット) で構成している。K-BUS はマイクロ命令で直接定数が指定できる。A-BUS、M-BUS は、インターフェースを介してメインメモリーのアドレス、データ各バスと結合される。

● マイクロ命令 マイクロ命令は、1命令50ビットで、演算指定、フラッグ操作指定、マイクロ命令の *next* アドレス指定、定数データ、内部バス制御等のフィールドから成る。内部バス制御フィールド (BCフィールド) は、6ビットで BCU を制御する。BCU は、各内部バス、S-MEM、CAR-CDR レジスタ、その他各種レジスタの制御及びメインメモリーの起動を行なう。マイクロ命令のサイクルタイムは通常 200 ns であるが、S-MEM をアクセスした時は 100 ns、メインメモリーをアクセスした時はメインメモリーのサイクル終了までクロックが延長される。

● スクラッチパッドメモリー (S-MEM) S-MEM は、20ビット 1K 語の構成でマイクロレベルのスタック及びワークエリアとして使用する。スタックとして用いるためにスタックポインタレジスタ (SPR) を備えており、スタック操作を行なった場合、SPR は自動的に増減される。S-MEM のアドレス指定は、SPR の他、A-BUS、M-BUS、K-BUS から行なう事ができる。S-MEM への書き込みデータは M-BUS、K-BUS から、読み出したデータは、M-BUS、A-BUS、WCS のアドレスへと種々の指定が可能で、非常に強力な使い方が出来る。読み出したデータをマイクロ命令のアドレスとすることが出来るため、本来 $\text{\textcircled{I}}3000$ シリーズにはなかった、マイクロレベルのサブルーチンコールが可能となった。S-MEM のスタックとしての機能は、インタプリタをマイクロレベルで記述するためには不可欠な機能である。また LISP のデータは、大部分がポインタ、即ち番地であるので S-MEM のデータ転送の相手として種々のバスを選択できるのは極めて有効であると思われる。

● メインメモリーとのデータ転送 EVAL プロセッサとメインメモリーとの間でデータの転送を行なう場合、A-BUS にアドレスを送出し、BCフィールドでメモリー操作を行なう事を指定する。データの転送は、M-BUS を通じて行なう。メインメモリーのアド

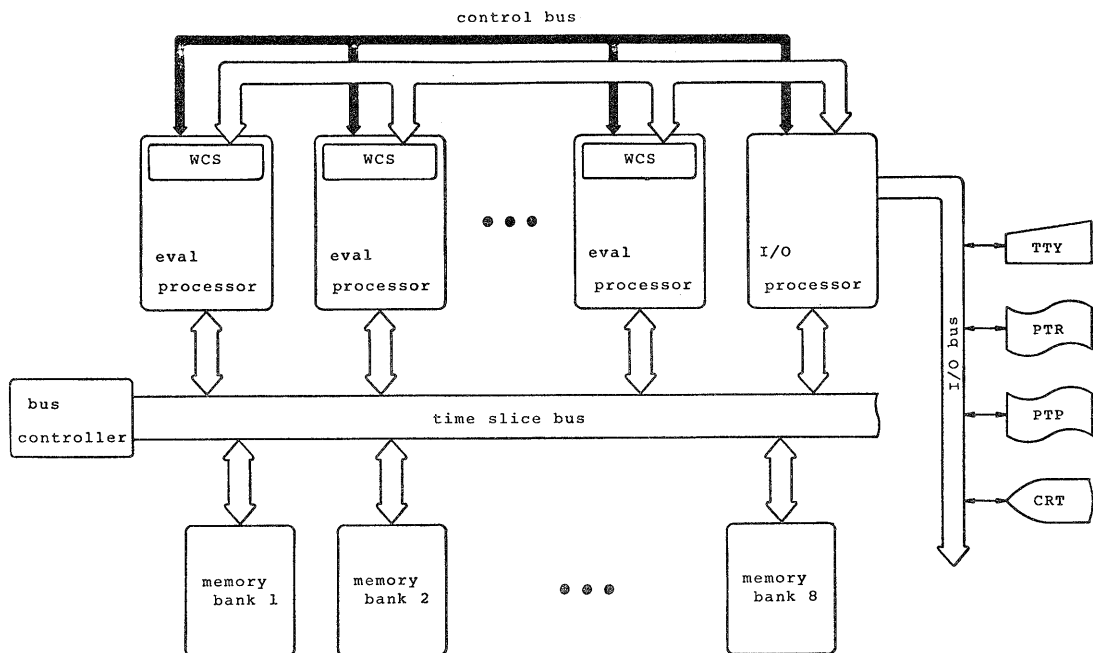


図3 EVLISマシンの構成

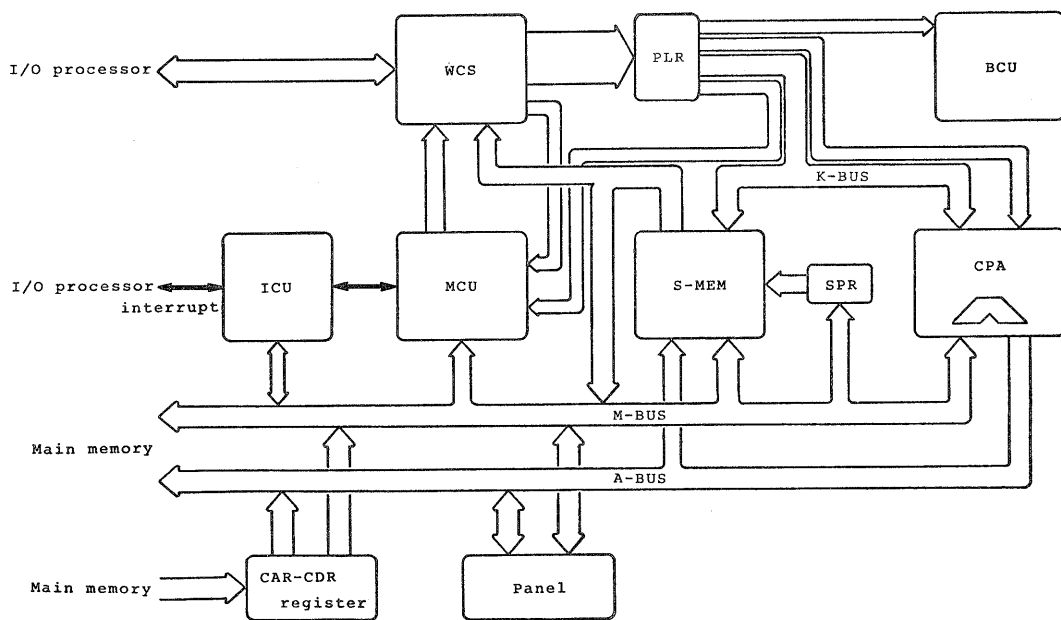


図4 EVALプロセッサの構成

レスは20ビット単位につけてあり、偶数番地は *car* 部、奇数番地は *cdr* 部に対応する。

● CAR・CDRレジスタ メインメモリーからデータを読み出した時は40ビット、即ち *car* 部、*cdr* 部両方が同時に得られる。M-BUS にはそのうち A-BUS で指定された片方の20ビットが乗せられるが、BC フィールドの指定によって、CAR-CDR レジスタに *car* 部、*cdr* 部を格納しておくことができる。CAR-CDR レジスタの内容は、M-BUS と A-BUS に出力する事が出来、この機能によって *caddr* 等は演算部を介さずに得ることができる。

3.3 I/Oプロセッサ

I/Oプロセッサは、 Σ 80マイクロプロセッサを用い、クロックサイクルは400ns、専用のローカルメモリーは、13K バイトである。その他、メインメモリー、WCS、入出力デバイス、割り込みのインターフェースを持つ。割り込みのインターフェースは、入出力デバイス、EVALプロセッサからの割り込み、及び EVALプロセッサへの割り込みを処理する。EVALプロセッサ相互の割り込みは全てI/Oプロセッサを介して行ない、I/Oプロセッサが管理する。

3.4 メインメモリーの構成

メインメモリーは、1語40ビット、4k語のバンク8個、計32k語で構成される。メインメモリー1語でリスト構造の1セルを表現し、*car* 部、*cdr* 部が同時にアクセスできる。バンク相互は、図3の様に1組のタイムスライスバスで結合し、バスコントローラーがバスを制御する。バスは1サイクル50nsで区切られており、プロセッサから見るとメモリーサイクルは、バス使用要求に1サイクル、バンクのアクセスに8サイクル、計450nsを必要とするが、メモリーから見れば、アドレス、データ転送のためにバスを使用するのは1サイクルのみであるので、異なるバンクは並列的にアクセス出来る。競合は、バスの競合とバンクの競合の2種の場合があり、前者の場合は1サイクルの待ち時間でメモリーサイクルが開始される。後者の場合にはそのバンクが開放されるまで待たされる。競合時におけるプロセッサのメインメモリーアクセスの優先権は、ラウンドロビン方式で設定される。プロセッサが、あるメモリーロケーションを排他的に操作したい場合は、凍結要求を伴うアクセス要求を出せば、そのロケーションの存在するバンクが凍結解除の指示があるまで、凍結要求を出したプロセッサに専有される。

4. シミュレーションによる評価

4.1 シミュレーションによる評価の対象

EVLISマシンの製作において、プロセッサの台数が高速化にどのような効果をもたらすかは興味のあるところである。また、複数台のプロセッサをどの *strategy* に従ってスケジュールするのが最もオーバーヘッドを少なくできるのか等を調べる必要がある。そのため EVLISマシンの並列的実行をシミュレートするシミュレータを製作して、その動きからそれらを比較・検討した。シミュレーションでの、並列的実行のための資源となる諸情報を、我々の研究室で既に製作・実用されている OLISP インタプリタに手を加えることにより、比較的容易に得られるので、OLISP を利用することにした。OLISPは、大型計算機 NEAC ACOS 900 上にインプリメントされているので、この実測値における各プロセスの長さは、その機械語1命令を最小単位として計測・表現する。

4.2 シミュレーションの概要

シミュレーションは、3段階に分けて行なった。

第1段階：シミュレーション時においては、プログラムを解釈する必要はなく、単にプロセスの *step* 数と、何個の子プロセスを生成するのかがわかればよい。プロセス長の計測のため、インタプリタを構成する各関数にその実行 *step* 数のカウンタを挿入した。プロセス間の関係を表わす情報を得るため、*evlis* ルーチンに次の機能を追加した。

- (1) a. *evlis* が *call* された時点で、その親プロセスの優先順位、引数の数(子プロセスの数)、そのプロセスの長さを組にして出力する。
 - b. プロセスが *value* を出した時点で、そのプロセスの優先順位、*value* を得たことを示す *flag*、そのプロセスの長さを組にして出力する。
- もし、副作用のある書き換えが、発生していれば、その位置をプロセスの長さと共に出力する。

- (2) a. b. それぞれに続いて、全てのカウンタを *reset* する。

これらのプロセスごとの情報をファイルに出力する。

第2段階：ファイル中の情報から直接シミュレーションを実施するのは、ファイルのI/Oを頻繁に行ない、非効率であるので、プロセスごとの情報を、有向グラフの形にしてメモリー内に組み立てる。その各 *node* は親プロセスが子を生じするまでの部分、子プロセス自身、親プロセスの残りの部分(それぞれをタスクと呼ぶ)に対応する情報と、各 *node* 間のリンクの情報で構成される。

第3段階：複数台のプロセッサを各 *strategy* に従ってスケジュールしながら、プログラムの実行をシミュレートする。その時のパラメータは、*strategy*、プロセッサ台数、切り替え時のオーバーヘッド(*stack frame*の作成・削除に関するもの)、書き換え時のオーバーヘッドである。シミュレーションは、プロセッサのどれかが、タスクの実行を終了した時刻で区切りながら行なう。プロセスは、その時刻において選択された *strategy* に従ってタスクを連結して構成する。従って、プロセスとして実行中であれば、プロセッサは実行を継続し、プロセスとしても実行終了であれば、プロセッサは解放される。プロセスのスケジューリングは *g-list* により管理する。オーバーヘッドはプロセス終了時に挿入する。副作用のある書き換えが発生していれば、プロセスのスケジューリング時に1つのタスクとして登録し、同様に扱う。それらの判定・処理が全て終われば、その時刻を出力し、次に終了するタスクを探す。このサイクルを繰り返す。

4.3 シミュレーションの結果

シミュレーションの結果として、除去不能の副作用を起こす書き換えがない場合のプロセッサ台数 n と切り替え時のオーバーヘッド α をパラメータとし、幾つかの特徴あるプログラム^[9]の実行における高速化の効果を表1に示す。

prog 形式のトップレベルにおいて、連続する関数 *setg* を並列に評価する関数を導入し、処理時間を短縮することができる。TPU-5にこの関数を適用し書き直したものが TPU-5' である。

SORT-20, WANG-B, TPU-5 は、処理時間に短縮の効果があるのは高々4台までである。これらのプログラムでは、その構成上、プロセスが並列的に発生せず、直列的に連結する性質を持ち、並列化の効果が上がらない。しかし、BIT-A-6の様に、関数の各引数(≥ 2)で、2個以上の引数を持つ関数を *call* (特に

表1. プログラムの処理時間 (stepで計測)

strategy 1

	$\alpha=0$ step					$\alpha=50$ steps					$\alpha=100$ steps				
	BIT- A-6	SORT -20	WANG -B	TPU -5	TPU -5'	BIT- A-6	SORT -20	WANG -B	TPU -5	TPU -5'	BIT- A-6	SORT -20	WANG -B	TPU -5	TPU -5'
1	431	819	54.5	2106	2160	—	—	—	—	—	—	—	—	—	—
2	218	685	42.3	1899	1619	277	769	51.1	1999	1773	338	852	60.1	2099	1927
3	150	674	39.7	1872	1542	193	757	46.7	1958	1661	238	840	53.9	2044	1782
4	116	674	38.9	1869	1510	150	757	44.5	1955	1619	184	840	50.7	2040	1728
5	95	674	38.6	1869	1503	122	757	43.9	1955	1609	151	840	49.4	2040	1716
6	81	674	38.6	1869	1501	105	757	43.9	1955	1605	129	840	49.1	2040	1709
7	72	674	38.6	1869	1501	93	757	43.9	1955	1605	114	840	49.1	2040	1709
8	65	674	38.6	1869	1498	83	757	43.9	1955	1601	93	840	49.1	2040	1704

($\times 10^3$ steps)

strategy 2

n	$\alpha=0$ step					$\alpha=50$ steps					$\alpha=100$ steps				
	BIT- A-6	SORT -20	WANG -B	TPU -5	TPU -5'	BIT- A-6	SORT -20	WANG -B	TPU -5	TPU -5'	BIT- A-6	SORT -20	WANG -B	TPU -5	TPU -5'
1	431	819	54.5	2106	2160	—	—	—	—	—	—	—	—	—	—
2	219	685	42.3	1900	1610	310	900	55.6	2197	1920	402	1115	68.9	2495	2228
3	151	674	39.9	1872	1540	212	879	51.5	2158	1820	274	1085	63.1	2443	2101
4	116	674	38.9	1869	1510	162	879	49.4	2154	1779	209	1085	59.9	2439	2048
5	95	674	38.6	1869	1503	133	879	48.9	2154	1769	172	1085	59.1	2439	2035
6	81	674	38.6	1869	1501	113	879	48.9	2154	1767	146	1085	59.1	2439	2033
7	72	674	38.6	1869	1501	100	879	48.9	2154	1767	128	1085	59.1	2439	2032
8	65	674	38.6	1869	1499	90	879	48.9	2154	1763	106	1085	59.1	2439	2027

($\times 10^3$ steps)

strategy 3

n	$\alpha=0$ step					$\alpha=50$ steps					$\alpha=100$ steps				
	BIT- A-6	SORT -20	WANG -B	TPU -5	TPU -5'	BIT- A-6	SORT -20	WANG -B	TPU -5	TPU -5'	BIT- A-6	SORT -20	WANG -B	TPU -5	TPU -5'
1	431	819	54.5	2106	2160	—	—	—	—	—	—	—	—	—	—
2	219	685	42.4	1900	1599	266	769	50.9	1999	1742	314	851	59.6	2099	1891
3	150	674	39.9	1872	1542	181	757	46.7	1958	1657	213	840	53.5	2044	1771
4	115	674	38.9	1869	1512	139	757	44.5	1955	1620	164	840	50.7	2040	1728
5	94	674	38.6	1869	1505	114	757	43.9	1954	1611	134	840	49.4	2040	1717
6	81	674	38.6	1869	1501	97	757	43.9	1954	1605	115	840	49.1	2040	1709
7	71	674	38.6	1869	1501	86	757	43.9	1954	1605	101	840	49.1	2040	1709
8	64	674	38.6	1869	1499	78	757	43.9	1954	1601	90	840	49.1	2040	1704

($\times 10^3$ steps)

strategy 4

n	$\alpha=0$ step					$\alpha=50$ steps					$\alpha=100$ steps				
	BIT- A-6	SORT -20	WANG -B	TPU -5	TPU -5'	BIT- A-6	SORT -20	WANG -B	TPU -5	TPU -5'	BIT- A-6	SORT -20	WANG -B	TPU -5	TPU -5'
1	431	819	54.5	2106	2160	—	—	—	—	—	—	—	—	—	—
2	220	685	42.4	1900	1598	245	750	48.1	1958	1689	271	822	54.3	2024	1785
3	149	674	39.9	1872	1541	167	747	43.9	1928	1617	184	822	48.7	1994	1699
4	115	674	38.9	1869	1511	128	747	42.4	1926	1585	142	822	46.8	1991	1666
5	94	674	38.6	1869	1505	106	747	41.4	1925	1578	117	822	44.9	1991	1659
6	81	674	38.6	1869	1501	90	747	41.4	1925	1575	101	822	44.6	1991	1657
7	71	674	38.6	1869	1501	80	747	41.4	1925	1574	88	822	44.6	1991	1656
8	63	674	38.6	1869	1499	73	747	41.4	1925	1573	79	822	44.6	1991	1656

($\times 10^3$ steps)

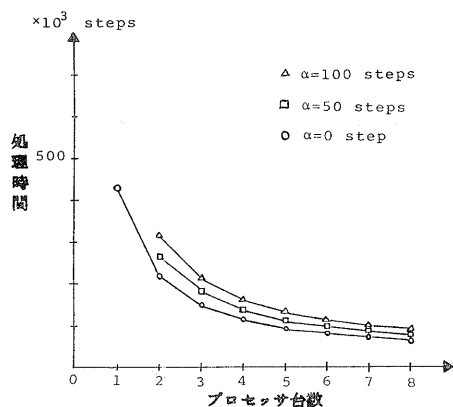


図5. strategy 3における
BIT-A-6の処理時間

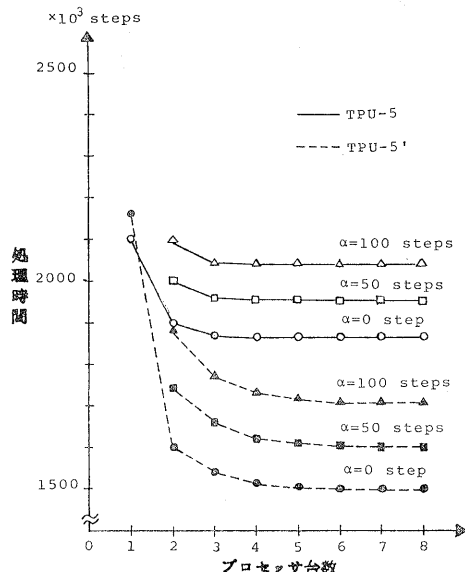


図6. strategy 3における
TPU-5とTPU-5'の処理時間

recursive call)する場合は、プロセスが並列的に発生し、その効果が観測できた。同様に TPU-5' でも、その効果が改善されていることがわかる。

表1の結果から、strategy 4 が最適と思われる。しかし、書き換えによる再評価を要する場合の結果が得られた時点で、strategy の選択を再考する必要がある。書き換えのある場合については現在データを採取中である。strategy 3 における BIT-A-6, TPU-5 と TPU-5' との比較を図5・図6に示す。今後は、さらに、オーバーヘッドの詳細な評価を加えると共に、EVLIS マシンの命令語によるシミュレーションを考えている。

参考文献

- [1] R. Greenblatt, "The Lisp Machine," MIT AI Lab. Working Paper, 79, 1974.
- [2] T. Knight, "CONS," MIT AI Lab. Working Paper, 80, 1974.
- [3] G. L. Steele Jr., D. A. Moon, "CADR," MIT AI memo, 1978.
- [4] 薄, 田丸, 所, "マルチプロセッサによるLISPマシンの試作," 情報処理学会 第19回全国大会講演論文集, 2B-7, 1978.
- [5] 日比野, "並列ガーベッジコレクタを組み込んだミニLISP," 同上, 2C-4, 1978.
- [6] 龍, 小林, 滝本, 多田, 金田, 前川, "試作LISPマシンとその評価," 情報処理学会 記号処理研究会, 記号処理 7-3, 1979.
- [7] 長尾, 中島, 伊藤, 川口, 三田村, "LISPマシンNK3のアーキテクチャとその性能評価," 同上, 記号処理 7-4, 1979.
- [8] 安井, 斎藤, 三石, 宮崎, "evlis の並列処理," 情報処理学会 第20回全国大会講演論文集, 3K-8, 1979.
- [9] 竹内, "LISP処理系コンテストの結果," 情報処理学会 記号処理研究会, 記号処理 5-3, 1978.