ファームウェアを利用したAPLインタプリタの構成法および評価

森本陽二郎
（東京芝浦電気(株)総合研究所）

1. はじめに

APLの特徴としては次のものがあげられる。

1) 変数の型が動的に変わる。
2) 演算子の種類が多い。
3) 配列演算を得意とする。
4) プログラムは演算子とオペランドの組み合わせにより構成され，構文が簡単である。

1)の理由により，実行直前までその演算の処理内容が決まらない場合が多い。このために，一般のコンピュータ言語にみられるコンパイラ方式は困難である。可能な方法としては，1行単位でコンパイルを試み，必要に応じてコンパイルし直す方法があるが〔9,10〕，小規模の計算機では，インタプリタ方式で処理するのが一般的である。〔4〕

マイクロプログラム方式の計算機では，ファームウェアを利用することにより，各言語処理に適したアーキテクチャを作り出すことが可能である。APLにおいてもファームウェアを利用することにより，高速のインタプリタの実現が試みられている。〔5,6,7,8〕

本報告は，EPOS（Experimental Polyprocessor System）上にファームウェアを利用して実現したAPL\EPOSIインタプリタについて，その処理方式およびファームウェア化による効果について述べる。

2. APL\EPOSIについて

EPOSは，ファームウェアが開放された計算機であり，ファームウェアを利用したインタプリタ方式により実現している。実行速度を速くするため，トランスレータが1行入力後，一旦内部形式（以下，中間形式と呼ぶ）に変換し，インタプリタが中間形式を解釈実行する方法をとっている。中間形式はソースプログラムへの逆変換可能な形であり，ソースプログラムは保持しない。

言語仕様はAPL\360とAPLSVの中間に位置する。

以下，マイクロプログラムをμpと記す。

2.1 中間形式

実行速度を速くするため，より低いレベルの中間形式に変換する必要がある。しかし，実行時に中間形式を作成し直すことを避けるためには，ソースプログラムの全てのシラブルに対応するコードを中間形式でも持つ必要がある。このため，ソースプログラムのシラブルを中間形式では，2バイト固定長のコードにより表現している。各シラブルはfig 1のように中間形式に変換される。また，1つのAPL関数に対応する中間形式をfig 2に示す。関数呼び出し，分岐等の処理を簡単にできるようにするため，いくつかの表を付加している。

中間形式とソースプログラムの大きさを比較した場合，プログラムの性質により多少異なるが，中間形式の方が約2倍の大きさである。この理由としては，1文字である演算子および記号に対し，2バイトのコードが生成され

ることと，実行時の数値変換をなくすため数値データについては，中間形式作成の段階で浮動小数点表示（４バイト）に変換するためである．

APL シラブル	中間形式
名前	名前表へのポインタ（名前表には名前が登録されている）
定数	定数領域へのポインタ（数値データは浮動小数点表現）
演算子	演算子コード（演算子は演算数の処理形態に基づいて分類されている）
記号	記号コード

fig 1　シラブルの中間形式

関数の属性（アーギュメント結果，行数，大きさ）
各行の中間形式（fig 1 にあげたシラブルの列）
行の番地表（分岐のため）
局所変数．ラベル表
定数領域

fig 2　APL関数の中間形式

2.2　EPOS ハードウェア

ハードウェアは，ポリプロセッサシステムEPOSの1つのコンピュータモジュールを使用している．各コンピュータモジュールには，

メインメモリ　128Kバイト
μpメモリ　16K語

が装備されている．マイクロ命令は，32ビットであり，垂直型に近い機能を有する．なお，EPOSは独自の機械語を持たない．〔11〕

ワーキングレジスタとして自由に使うことのできるレジスタは40個ある（各16ビット）．うちわけは，ジェネラルレジスタ（GR）4個，レジスタファイル（RF0，RF1）16個×2，カウンタレジスタ（CTR）2個，インタフェスレジスタ（IFR）2個である．

μpで記述されたOSの下で稼動することを考慮し，μpメモリへの書き込みを禁止した．メインメモリはAPLユーザ領域と，APLプロセッサのうちμp以外の部分をおく2つのセグメントに分割し，後者には書き込みに対する保護を施してある．なお，μpメモリには，OSの常駐部が入るため，できるだけ少ないμp量でAPLを実現するよう要求されている．

このようなハードウェアに対し，システム変数割りつけにはfig 3のような階層をつくることが可能である．参照の頻繁な変数から順に，参照に必要なマシンサイクルが少なく，命令数の少ないハードウェアを割り当てる．例えば，浮動小数点演算に必要な変数は，RF0，GRを用い，構文解析用のスタックの先頭へのポインタはRF0の1個を割り当てている．メインメモリにあるワーキングデータ領域のうち空き領域の先頭を指すポインタはRF1の

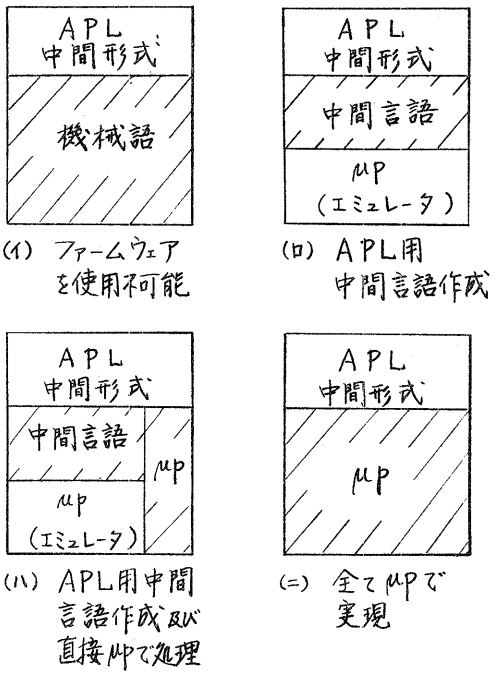
1個を使用する．μPメモリは書き込みを禁じられているが，デシジョンテーブル等参照の頻繁な読み出し専用のテーブルをおくことができる．RF0の内容を保存する必要があるときは，RF1あるいはメインメモリに保存している．

ハードウェア		データ長（ビット）	参照に要する命令数	マシンサイクル数
GR，IFR RF0，CTR		16	1	1
RF1		16	3	3
μPメモリ		32	3	5
メインメモリ	整数	16	3	11
	文字	8	5	13
	実数	32	4	18
	配列	要素数によって異なる		

fig 3　システム変数に使用されるハードウェアの階層

2.3　APL\360インタプリタについて

インタプリタの実装方法は，使用可能なμPメモリの容量とマイクロ命令の機能に大きく依存する．主な実装方法の例をfig 4に示す．インタプリタ全てをμPで記述できるほど大きなμPメモリを有する計算機では全てをμPで記述する（fig 4（ニ））であろうが，EPOSの場合はμPメモリ容量の制限からμP記述部分の選択が必要である．即ち，fig 4の（ロ）あるいは（ハ）の方法で実現するのが適当である．インタプリタのうちで，使用回数の頻繁な部分，例えば，中間形式の走査や基本的な演算については最も高速で処理することが必要であるため，直接μPで記述することが望ましい．したがって，APL\360では（ハ）の方式を採用している．



（イ）ファームウェアを使用不可能　（ロ）APL用中間言語作成　（ハ）APL用中間言語作成及び直接μPで処理　（ニ）全てμPで実現

fig 4　インタプリタ実装法の例
（斜線部はインタプリタ）

また，エミュレータ作成の作業を省くため，中間言語としては擬似APLを用いた．擬似APLを以下PAPLと呼ぶ．

PAPLはAPLと同じ文法を持つが，使用できる演算子が限られている．オペランドはスカラデータが主で，そのうちのほとんどの場合直接μPが処理するようになっている．PAPLは中間形式となってシステム内に存在する．

インタプリタの主な機能を次にあげる。

1) 中間形式の走査，演算単位（演算すべき演算子とオペランドの組み）の決定
2) 演算単位の実行
   a) 演算数内の要素の取り出し制御
   b) 基本演算（四則演算等）

配列演算では，2)に比して1)の実行時間は少ないが，演算単位の内容を問わず常に実行しなければならない部分である。したがって，システムを総体的に速くするためにμPで記述している。2)はa)とb)の組み合わせにより処理される。b)のほとんどはμPで記述しているが，a)については，実際の取り出しのみμP化されており，そのための指標の計算はPAPLで行なう。また，オペランドの属性に対する演算の正当性の検査もPAPLで行なう。

μPで記述した主な部分の大きさはスカラの基本演算（836語），配列処理（2200語），特殊な演算子（← →，単項ㄗ ⍴等1262語），APLおよびPAPLの走査，制御の受け渡し，関数呼び出し（1919語）である。他にトランスレータ，逆変換，編集修正に必要な部分，I/O，ファイル処理を含めると合計約9000ステップに及ぶ。μP以外では記述できない部分や，最も高速で処理する必要のある部分のみをμPで記述している。それ以外全てPAPLで記述しており，トランスレータ，逆変換，編集修正の主な部分もPAPLで記述している。インタプリタ部に必要なPAPLは，700ステップである。PAPLは，APL中間形式と同じ形でシステム内に存在し，1ステップ平均20バイトである。

PAPLの例をfig 5にあげる。図中，下線の関数および演算子はμPにより直接処理される。Aから始まる名前はPAPL関数である。その他の名前は変数である。AFDDTは基本的な二項演算子（+ − × ÷ = ≠ < ≤ > ≥ ∨ ∧ ⍱ ⍲ ⌊ ⌈ ｜）を処理する関数であり，スカラどうしに還元された演算を第10行で行なう。即ち，インタプリタが演算子コードを記憶した後，AFDDTに制御を移す。次に第10行を実行するときに記憶していた演算子コードを◎にあてはめて基本演算を行なう。また，8～11行の間を次元数だけループすることにより，配列演算が処理される。

```
∇V←LARG AFDDT RARG
[1]  1 AFFORM LARG        ／＊ 左オペランドの検査
[2]  2 AFFORM RARG        ／＊ 右オペランドの検査
[3]  AFCHK CHKTB1         ／＊ 演算の検査
[4]  U1←,LARG             ／＊ ベクトル化
[5]  U2←,RARG             ／＊ ベクトル化
[6]  I←0                  ／＊ カウンタ
[7]  V←MAX1⍴0             ／＊ 結果の領域
[8]  →(I XEQ MAX1) COMPS 12
[9]  I←I XPLUS 1
[10] V[I]←U1[I] ◎ U2[I]   ／＊ スカラ演算
[11] →8
[12] AFDEND CHKTB1        ／＊ 結果の変形
[13] ∇
```

fig 5 PAPLの例

3. ファームウェア化による効果

3.1 μP化率の定義

ファームウェア化による効果の基準は，「μP量に対する速度の向上する割合」である。しかし，μPは各計算機によってマイクロ命令の機能にかな

りの差があり，またμPルーチンの作成方法により同じ機能をμP化した場合でもμP量は大きく異なる．同じμP量でもμP化する部分の違いによっては，ある演算子では速いが別の演算子では遅くなるようなことが起こる．したがって，一般的なデータを求めるのは難しい．

APL\EPOSIでは，稼動に必要な最小部分をμPで記述している．ここでは，PAPLの部分を選択してμP化することによりその効果を調べる．μP化は，ある程度まとまった機能を持っている行単位，あるいは関数単位で行なうことができる．μP化の目安として，μP化率γを次のように定義する．

$$\gamma = \frac{\text{PAPLのモジュールMのうちでμP化された部分の大きさ}}{\text{PAPLモジュールMの大きさ}}$$

$0 \leq \gamma \leq 1$であり，当初のAPL\EPOSIは$\gamma = 0$である．また，γに対する速度向上の度合いを

$$t = \frac{\text{μP化率}\gamma\text{のときの実行時間}}{\text{μP化率}0\text{のときの実行時間}} \times 100$$

で表す．

3.2 簡単な演算子を実行したときの効果

A◎B（◎は＋－×÷＝≠＜≤＞≥∨∧⍱⍲⌊⌈）を処理するためのPAPLで記述された関数をμPで記述した．それらの関数の構造と大きさをfig 6に示す．（AFDDTはfig 5参照）これら全部をμP化した場合，即ち　$\gamma = 1$のときA＋Bの実行には

$118n + 531$　（nは次元数）

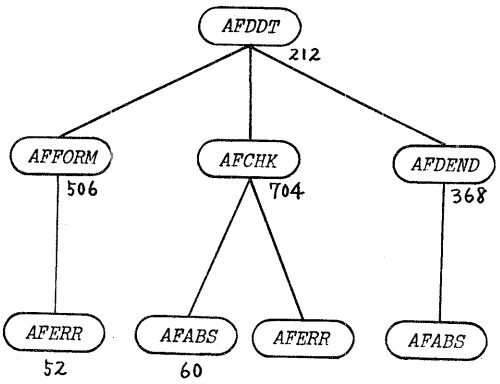


fig 6　A◎B（◎は基本演算子）を処理するPAPLで書かれた関数の構造と各関数の大きさ（単位バイト）

マシンサイクル要する．一方，μP化しないとき，即ち，$\gamma = 0$のときは

$11839n + 105137$

マシンサイクル要する．他の基本演算子の場合にも同様の結果が得られる．この例は極端な場合であるが，100倍以上速度に差がある．$\gamma = 1$のときμPは231ステップであり，$\gamma = 0$のときMは1902バイトであった．また，実行されるPAPLは

$4n + 36$行

である．

◎A，A◎B，◎/A，◎\A等の演算を行なった結果，PAPL1行あたりの平均実行時間は2900マシンサイクルであり，例外的なものを除いて意外に平均している．複雑な処理を必要とする演算子についても同様のことがいえる．このことをもとにして，A◎Bについて関数単位でμP化することによる速度向上の度合いをfig 7に示す．次元が大きくなるにつれ効果があがる．この例では，まず最初にAFDDTをμP化すると最も効率が良く

なり100次元のオペランドではr=0.11のとき t=6.19 となる。即ち，10％のμP化により10倍以上速くなることがわかる。これは，実際の演算部であるAFDDTの8～11行（fig 5参照）がμP化されたことによるものである。また，fig 7はμP化する部分の選択を誤るとほとんど効果があがらないことを示している。

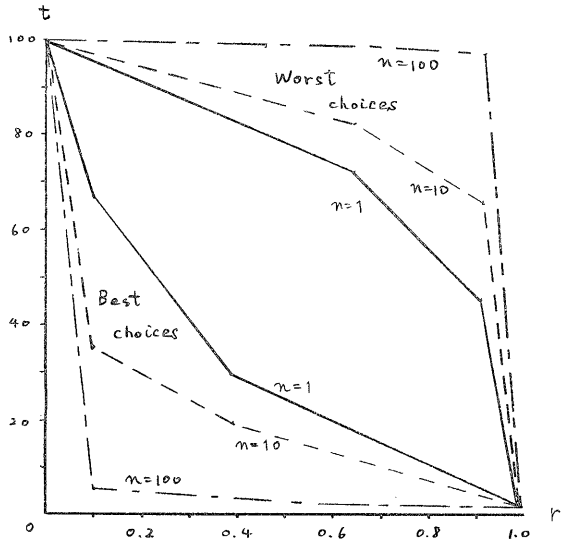
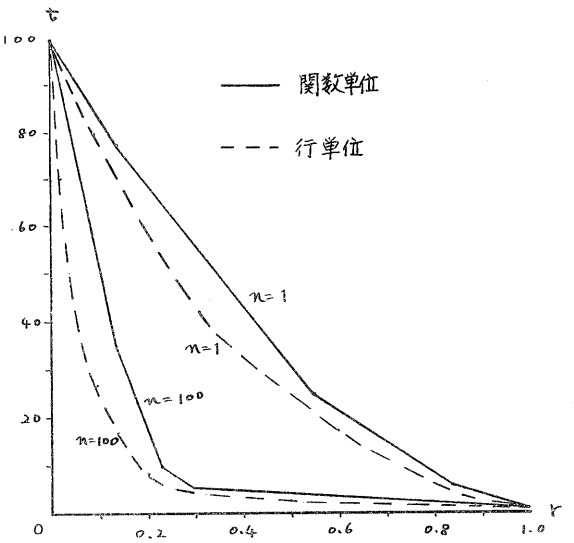


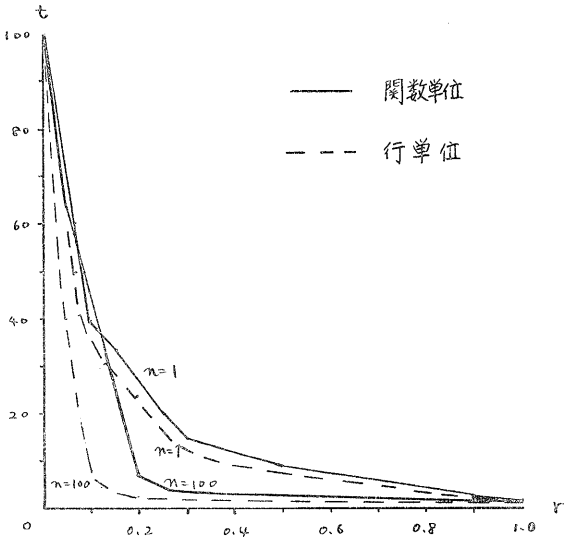
fig 7　A ◎ Bの速度向上
（◎は基本演算子）

3.3　多くの演算子を実行したときの効果

基本演算子，およびほとんどのAPL演算子を各1回実行したときのμP化の効果をfig 8に示す。オペランドの次元は1および100である。関数単位でμP化した場合を実線，行単位でμP化した場合を破線で示す。ただし，後者ではμP化された行で参照されていた変数を他のμP化されていない行で参照しているとき，変数の型が一致しないことがある。その変換が必要と



(a) 基本演算子に関する速度の向上
単項演算子　＋－×÷～L⌐|
二項演算子　＋－×÷＝≠＜≤
＞≥∨∧⍱⍲L⌐|



(b) APL演算子に関する速度向上
演算子は基本演算子，および
⍋ ⍒ ⌽ ⊖ ⌿ ⍀ ⍉ ⊥ ⊤ / \
∈ ↑ ↓ ⊥ ⊤ ⍟.⊗ ∘.⍟
（⍟ ⊗は基本演算子）

fig 8 各演算子を1回実行したときの速度向上

なるため実際にはfig 8より多少効率が悪くなる。

(α)より(β)の方が効率が良くなるのは，基本演算子以外の演算では多重ループが多く存在することと，特定の基本演算を多く行なうことによるものである。

3.4 簡単な関数を実行したときの効果

2つの簡単なAPL関数について，MP化したときの効果を調べる。一方はデータを入力して昇順にソートする関数（SORT）で，他方は sinh を求める関数（SINH）である（fig 9参照）。前者は，A[⍋A]，後者は5○×と同じ機能を果す。

```
    ∇R←SORT A;B;C;D
[1] B←⍴,A←⍬
[2] R←⍳0
[3] D←⍴,A
[4] A←(A≠C←⌊/A)/A
[5] R←R,(D-⍴,A)⍴C
[6] →(B≠⍴R)/3
[7] ∇
```

(α) 昇順にソート

```
    ∇R←SINH X;N;Y
[1] N←1+R←0
[2] →(R=Y←R+(X*N)÷!N)/0
[3] R←Y
[4] N←N+2
[5] →2
[6] ∇
```

(β) hyperbolic sin

fig 9 APL関数の例

SORTの場合，PAPLは（19n² + 523n）/2行（nはデータの数），SINHは，1475行実行される。両者とも同じ9つのPAPL関数を使用している。これらの関数を最も効率の良いものから選択してMP化した場合の速度向上の度合いをfig 10に示す。また，一方を最も効率良くMP化したときの他方の効果を示してある。最も頻繁に実行される部分から約10%までは両者同じ場所であることがわかる。なお，SORTについては，データ数10の場合のデータである。両者を通じて効果の上がり方が良くないのはスカラ演算が主となっているためである。

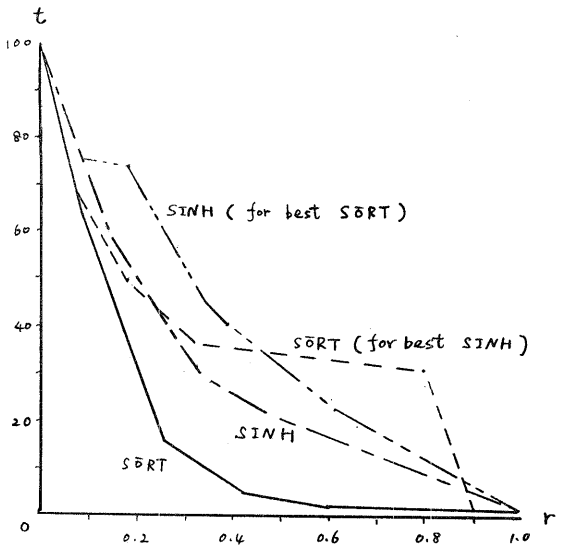


fig 10 2つのAPL関数についての速度向上の度合い

3.5 ファームウェア化インタプリタとコンパイラの比較

APLプログラムを一般的な機械語に変換するコンパイラの存在を仮定し，そのオブジェクトを実行した場合とファ

ームウェア化インタプリタが実行した場合を比較する．

例えば，A＋B（A，Bはn次元）というAPL演算をコンパイルするとfig 11のようになる。これは最も効果的にコンパイルされた例である。機械語としては，ミニコンTOSBAC-40の機械語を採用した。この機械語は他の計算機に比して標準的なものと考えられる。EPOSの場合でも汎用の機械語を作成すれば，このレベルのものになるであろう。

fig 11の命令コードをEPOSのMPでエミュレートして実行させると

命令フェッチ　115n＋7
デコード　73n＋7
各命令実行　309n＋10（最低）

合計すると

497n＋24マシンサイクル要する。他の機械語の場合でも，APL用に作成されない限り，この数値に大差はないと思われる。

r＝1におけるA＋Bの速度と比較すると，n≧2では，ファームウェア化インタプリタの方がコンパイラより速い。また，n＝100の場合を例にとるとr＝0.25でほぼ同じ速度になりr＞0.25ではインタプリタの方が速い。オペランドの次元数が大きい場合，他の演算子でも同様のことがいえる。

4．おわりに

APLプロセッサの構成法とファームウェアの利用による効果をAPL\EPOSIについて述べた。

ファームウェアにより100倍以上も速くなることや，一般的な機械語へのコンパイラよりも速くなりうるほどファームウェアの効果は大きい。また，演算数の次元が大きい程，効果が顕著に現われる。

```
        LIS     R5,0        /* カウンタ
LOOP    LE      F0,DA(R2)   /* Aの要素
        LE      F4,DA(R3)   /* Bの要素
        AE      F0,F4       /* 加算
        STE     F0,DA(R4)   /* 結果
        AIS     R2,4        /* Aの要素の番地
        AIS     R3,4        /* Bの要素の番地
        AIS     R4,4        /* 結果の格納番地
        AIS     R5,1        /* カウンタ
        CHR     R1,R5       /* R1は次元数
        BNE     LOOP        /* ループ
        BR      R15         /* 終了
```

fig 11　A＋Bのコンパイルリスト

今後，演算子の使用頻度も考慮に入れた高速化の検討も必要である。なお，本研究は通産省大型プロジェクト「パタン情報処理」の一環として行なわれたものである。

5．謝辞

本研究に関してご指導いただいた東大　前川　守助教授，データの収集整理に協力していただいた芝浦システム(株)小田喜久栄嬢　また，ハードウェアモニタによるデータ収集に協力していただいた東芝総研　神谷茂雄氏に深謝します。

参考文献

[1] K. E. Iverson：A Programming Language，John Wiley & Sons. Inc.，1962

〔2〕 IBM manual : APL\360 Primer
〔3〕 IBM manual : IBM 5100 Reference Manual
〔4〕 渡辺豊英他 : ミニコンにおけるAPL会話型処理システム，情報処理，Vol. 16，No. 9，1975
〔5〕 宮脇富士夫他 : APL会話型処理システムのインタプリタの分析とファームウェア化の要点，情報処理，Vol. 19，No. 5，1978
〔6〕 宮脇富士夫他 : APLインタプリタのファームウェア化とその効果について，情報処理論文誌，March，1979
〔7〕 Hassit, A., Lageschulte, J. W., and Lyon, L. E. : Implementation of a high level language machine, CACM 16, 4, April, 1973
〔8〕 Zaks, R., Steingart, D., and Moore, J. : A Firmware APL time-sharing system, AFIP Conf. Proc., Vol. 38, 1971 SJCC
〔9〕 Eric J. van Dyke : A Dynamic Incremental Compiler for an Interpretive Language, Hewlett-Packard Journal, July 1977
〔10〕 Leo J. Guibas : Compilation and Delayed Evaluation in APL, Fifth Annual ACM Symposium on Principles of Programming Languages January, 1978
〔11〕 Maekawa, M., et al. : Experimental Polyprocessor System (EPOS)－Its Processors, Proc. International symposium on Computer Architecture, Philadelphia, April, 1979
〔12〕 Maekawa, M. and Morimoto, Y : Performance Adjustment of An APL Interpreter, Proc. EUROMICRO 79, August 1979