ICOT Technical Memorandum: TM-0209

TM-0209

Prolog マシン

横田　実

August, 1986

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191－5
Telex ICOT J32964

Institute for New Generation Computer Technology

Ｐｒｏｌｏｇマシン

Ｐｒｏｌｏｇ　Ｍａｃｈｉｎｅ

横田　実
Minoru Yokota

日本電気（株）　Ｃ＆Ｃシステム研究所
NEC Corporation

１．はじめに

論理型言語Prologは論理学をベースにしたプログラミング言語として1970年代の初めに誕生した。従って、その歴史はもう10年以上になる。しかしながら、Prologのための専用マシンの開発が開始されたのはごく最近のことである。それまでは自然言語処理研究のためのプログラミング言語として主にヨーロッパを中心に比較的少数の研究者の間で利用されていたにすぎない。1982年に始った日本の第五世代コンピュータシステム研究開発プロジェクトがPrologを研究開発の基本言語として採用し、そのための専用マシンの開発を開始したことから世界的な注目を浴びることになった。本論文ではPrologマシンの開発動向とマシンアーキテクチャについて主に逐次型Prologマシンに焦点を絞って要約する。

２．Prologマシンの開発状況

Prologマシンの研究開発はLispマシンの場合と似通った経過を辿っているが、より早いペースで進みつつある。即ち、Lispマシンの開発が汎用計算機上での十分な経験を積んだ後にようやく行われたのに対して、Prologマシンの場合は逆に専用マシンの開発によってPrologの利用範囲を拡大しつつある。

ユーザ言語として考えた場合、Prologは非常に使い易いプログラミング言語といえるが、これまで実行効率が悪い上にメモリを大量に消費するため応用プログラムを書く人達からは実用的でないと考えられてきた。しかし、Prologマシンの出現により論理型言語のもつ利点を実用的レベルで活かすことが可能になりつつある。[1,2] 特にICOTで開発されたパーソナル逐次型推論マシンPSIは実用的ツールとして利用することを最初から意図して作られた本格的なPrologマシンであり、現在約60台ほどがICOTおよびその周辺で利用されている。また、各国で進めている第五世代コンピュータ研究開発プロジェクトでも同様のPrologマシンの開発が計画されている。また、1983年にD.H.D. Warrenにより新しいPrologマシン命令セットとコンパイラの最適化技法が提案されたことによって汎用計算機上でも比較的高速の処理系を実現することが可能になった。[3]

Prologマシンの対象とするProlog言語仕様としてDEC-10Prologが世界的標準になっているが実用的プログラミング言語としてみた場合に十分とは言えず、いくつかの機能強化が試みられている。例えば、PSIの場合には実行順序制御機能の強化とオブジェクト指向の概念の導入がおこなわれている。また、実行速度を上げるために決定的な処理を記述できるような制御構造やデータタイプ宣言などの導入も提案されている。

表１にこれまでに開発された逐次型Prologマシンの諸元を示す。Prologマシンの実現方式としては単なる言語処理プロセッサとして位置付ける方式とPrologを実行するための専用システムとして開発する方式に分けられる。前者の場合はシステムとして利用するためにオペレーティングシステムとのリンクが必要となり、何等かのホストマシンの存在が要求される。ホストマシンとの結合の仕方はメモリを共有し合うCo-Processor方式とホストマシンと分離したBackend-Processor方式とが考えられる。[4] カリフォルニア大バークレイ校のPLM[5]や神戸大のPEK[6]は前者の方式、ICOTで開発されたCHI[7]は後者の例である。Prologシステムとして単独で利用可能とするには入出力等のオペレーティングシステム機能をPrologマシンの上に実現することが必要となる。PSIでは単一言語環境を実現するためにPrologにシステム記述の機能を導入し全てのソフトウェアを論理型言語で記述する方針を採用している。

３．Prologの実行メカニズム

Prologプログラムは事実もしくはルールを記述したデータ

ベースとみなせる。プログラムの実行はこのデータベースに対する問い合せを実行することに他ならない。問い合せの実行には次のような2種のメカニズムがある。まず問い合せに対して全く同一の事実が定義されているかを調べることである。ここで、”同一”とは同じ述語名、同じ引数をもつことであり、引数が変数の場合には”同一”にするために変数の書き換え（即ち、変数への代入）が行われる。次の例では変数Whoをjohnに置き換えることにより同一化できる。この同一化操作はユニフィケーションと呼ばれ、ユニファイ可能な事実が定義されていれば問い合せは”真”であるとして完了する。

```
?- father(Who, tom).

father(john, tom).
```

問い合せに対するもうひとつの実行メカニズムは”同一”の述語名、引数を左辺に持つルールが定義されている場合である。そのようなルールは問い合せを満たす可能性をもっているので、更に、右辺の述語（ゴールと呼ばれる）が真かを調べることが必要となる。次の例では述語fatherを左辺に持つルールに含まれる変数FatherをWhoに、Childをtomに置き換えれば同一にすることができる。

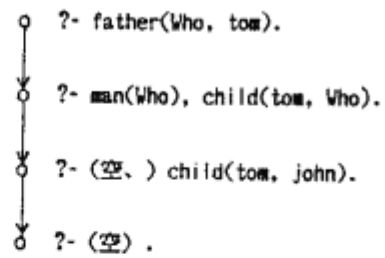
```
?- father(Who, tom).

father(Father, Child) :-
        man(Father), child(Child, Father).

man(john).
child(tom, john).
```

この結果、最初の問い合せはルールの右辺を書き直して次のように問い合せるのと全く同じになる。

```
?- man(Who), child(tom, Who).
```

この例では述語man, childが事実として定義されているのでWho = johnの書き換えを行うことにより初めの問い合せが真となる。

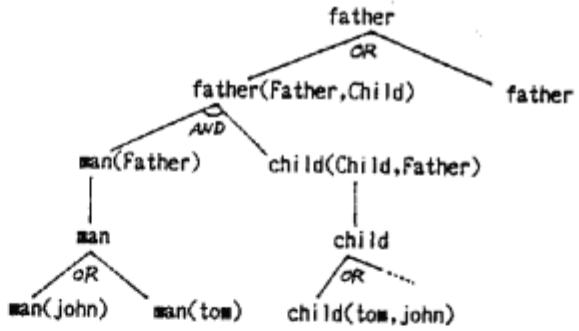
”事実”の定義も右辺が空のルールとみなせば、Prologの実行とは最初に投げられた問い合せを、ユニファイ可能なルールの右辺で置き換えることにより次々と変形していく操作になる。この操作により初めの問い合せが”空”に変形できれば実行は完了する。このような”書き換え”によって計算が実現される計算モデルはリダクションと呼ばれる。



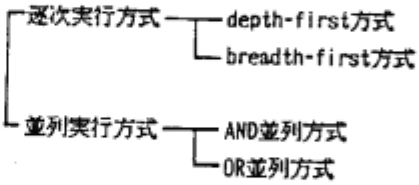
このような書き換えを行う際に、ユニファイ可能なルールが2つ以上あればとりあえずそのうちのひとつを選び、その右辺が真かのチェックを続ける。もし、この選択が誤りであった場合には元に戻って別のルールを適用し直す必要がある。これがバックトラックと呼ばれる後戻り処理である。

4．ゴールの評価順序

上記例ではゴールの書き換えを左から順に適用しているが、適用順序によりいくつかの方式が考えられる。ひとつのルールはANDで結合した複数のゴールを右辺に持ち得る。また同一の述語を左辺に持つ複数のルールはORで結合しているとみなせるので、Prologプログラムの実行空間は一般にAND-ORトリー形式で表現できる。



ゴールの評価順序とはこのAND-ORトリーの辿り方であり、次の4通りの評価方式がある。



1）逐次型Prologマシン

ゴールの書き換えをAND-ORトリーの最も左の枝から順に深さ方向優先で評価していく方式を(left-most)depth-first方式と呼ぶ。この方式は可能な選択肢のうちの一本だけを逐次的に評価してゆくものであり、実行順序を制御するために保存しなければならない情報量は比較的少なくメカニズムも単

純である。但し、選んだ選択肢が誤りであった場合には別の選択肢を選び直すバックトラック処理が必要となる。この方式の欠点は深さ方向を優先するためループ状の枝が存在する場合、実行が停止しないことである。

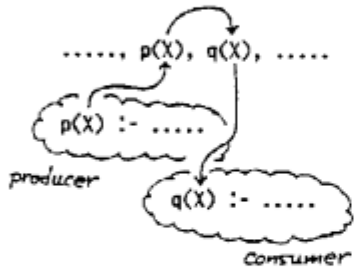
これに対して、AND-ORトリーの各枝を並行に評価していく方式をbreadth-first方式と呼ぶ。この方式は可能な選択肢を並行に評価してゆくので上記の欠点は無く、解があれば必ず停止する。しかし、複数の選択肢を同時に評価するためには多数の枝に関する途中情報を管理する必要があり、実行メカニズムが複雑化しあまり現実的な方式とはいえない。

従って、逐次型Prologマシンとしては前者のdepth-first方式を用いるのが一般的であり、AND-ORトリーを辿るのにスタックを用いるのが普通である。

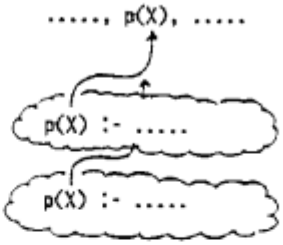
2) 並列型Prologマシン

Prologのセマンティックスは元々ゴールの評価順序には依存しない。従ってANDもしくはORで結合したゴールをどの様な順序で実行してもよい。ANDで結合したゴールを並列に評価することをAND並列、ORで結合したルールを並列に評価することをOR並列とそれぞれ分けて呼ばれる。

AND並列の場合にはゴール間で共有されている変数の扱いが問題となる。即ち、2つ以上のゴールが並列に評価されることにより同一の変数が同時に異なる値に書き換えられる事態が生じ、その正当性のチェック (consistency check)が要求される。変数の書き換えはまったく非同期に行われるので正当性チェックを効率良く実現することは難しい。但し、次の例のように変数への値の代入を一方向に制限することによりこのチェックのコストを小さくすることが可能である。これはストリーム型データを用いたproducer-consumer問題の記述を可能とする。



OR並列の場合にも並列に評価されるルールが同一の変数に対して異なる値を代入しようとすることが問題になる。この場合も呼出し元へ渡す値を一本のストリームにマージして渡すことが考えられる。



Prologの並列実行はこの他にもシステム全体の制御、カットのような選択肢を制限するメタな機能の実現、あるいはサイドイフェクトを伴うような逐次性を必要とする機能の実現などにまだ解決すべき問題があり、比較的ピュアなProlog処理系を対象とするのに留まっている。

いずれにしてもゴールの評価はデータ駆動により起動されることになる。従って、pure-Prologの場合にはデータフローマシンとの相性が良い。また、同一変数への複数の値の代入はストリーム型のデータを用いて実現されるのが一般的である。並列型Prologマシンでは処理の逐次性を前提にできないためスタックのような機構は利用しにくく、むしろゴールの書き換えを基本とするようなリダクションモデルに基づく実行方式が選ばれており、逐次型Prologマシンとは大分様子が異なる。

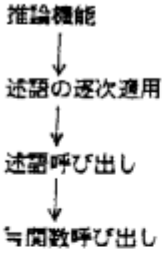
並列に評価する単位の大きさによっても実現の方式は異なってくる。引数単位の並列ユニフィケーションというような可能な限りの並列性を引き出そうとすると、いわゆるデータフローマシンに近づく。逆に、ある程度まとまったモジュール単位での並列性を引き出す場合には逐次型Prologマシンを要素プロセッサとするようなマルチプロセッサ方式に近づくであろう。

並列型Prologマシンについてはまだ研究段階であり、対象とする論理型言語についても並列処理記述言語としての見直しが活発に行われている段階である。本論では以降、逐次型Prologマシンのアーキテクチャについて述べる。

5. 逐次型Prologマシン

Prologマシンの高速化におけるキーポイントは実行順序制御とユニフィケーションであるが、特に逐次処理ではdepth-first評価方式を採用することからバックトラック処理が必要となり、実行順序制御の方式が高速化の鍵となる。

Prologの実行モデルは基本的に"書き換え"であることを既に述べたが、これを忠実に実現しようとすると文字列処理となり高速の実行は困難となる。従って、スタックを用いた通常の関数（もしくは手続き）呼出しとして実現するのが一般的である。



そこでは呼出すべき述語への引数の渡し方、バックトラックに備えた環境の保存の方式が重要である。

1) スタック

関数や手続きの呼出しを実現する場合、呼出元への戻りアドレスを記憶するためにスタック（コントロールスタック）が用いられる。また関数の持つ局所変数を動的に生成するためにもスタック（ローカルスタック）が必要となる。Prologにおける述語呼出しも関数の呼出しと基本的には同じであり、同様なスタック機構が適用できる。

通常の関数の実行は決定的に完了するため呼出し元へリターンする場合にはスタックに格納した戻り先アドレスを取り出した後、呼出し時に生成した全ての情報を捨ててしまって良い。ところが、Prologの実行ではバックトラックを実現するために環境の保存が必要であり、スタックをたたむことができない。更に、別の選択肢を評価するためには初めに呼出した時と全く同じ状況を再現できなければならず、前の選択により変数への代入が発生した場合にはもとの状態に戻すメカニズム（トレイルスタック）が必要となる。

このような幾つかのスタックへのデータの格納、管理がPrologマシンでの基本処理となる。

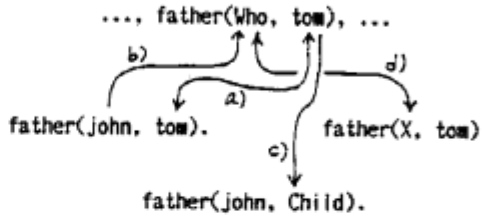
述語の呼出しからリターンしてもスタックをたためないということは呼出し元と呼ばれた側の環境が必ずしも連続しているとは限らず、スタックへのアクセスが不連続となることを意味する。従って、これまでの計算機で用いられていたスタック先頭部分だけを保持するような単純なスタック機構は有効でなくなる。結果としてPrologの述語呼出し機構は関数型の言語に比べて重くなり、また実行に必要となるメモリ量も多くなる。Prologが動的に構造体データを生成できる（グローバルスタック）こともメモリ消費量の増大を招いている原因の一つとなっている。

2) ユニフィケーション

ユニフィケーションは概念的には引数間のパターンマッチング処理と考えれば良いが、次の点に特徴がある。

a) マッチングによる変数への代入方向が双方向である。
b) 双方の変数が未定義の状態でも同一化がはかれる。

ユニフィケーションにおける４つの処理タイプを次の例で示す。



a) 比較による一致チェック
b) 呼び出し元変数への代入
c) 呼ばれた側変数への代入
d) 未定義変数同士の同一化

特にd)の場合がPrologに特徴的であり、２つの変数が同一であることを保証するために変数セル間をポインタで結ぶ処理が行われる。この時、ポインタの向きも考慮する必要があり変数セルアドレスの比較が必要となる。

更に、b),c),及びd)の場合にはバックトラック時に備えて代入の行われた変数セルアドレスを記憶することが必要となる。このためにスタックを用いるのが普通である。

Prologには変数に対する”データ型宣言”がないため実行時の動的なデータタイプチェックが不可欠である。ユフィケーションにおける変数値の比較やデータタイプによる処理の分類を高速化するにはメモリの各語にタグを設けるのが有効である（タグアーキテクチャ）。また、変数へのアクセスを高速化するためにはここでもスタックやメモリアクセスの高速化が重要となる。

3) 引数渡しの方法

ユニフィケーションは呼び出し側ゴール引数の値と呼ばれた側ルール左辺の引数の値との比較を行うことになる。最も単純な方式は呼び出し側と呼ばれた側双方のスタックフレームを直接アクセスすることである。この方式では変数へのアクセスが常にメモリアクセスとなること、バックトラックにより他の選択肢を試みる場合に呼び出し側引数の値を再度評価しなければならないこと等により、効率の良い実現が難しい。

述語の呼び出し処理を普通の関数呼び出しと考えれば、ユニフィケーションに必要な引数の値を呼び出された側に引き渡してしまう（コピーする）方式が考えられる。呼び出された側の引数が例えば次のような未定義変数の場合には直接、呼び出し元引数の値を代入しても構わない。

```
father(X,Y) :- .....
```

これは呼び出し側引数のコピー先を呼ばれた側変数セル領域にオーバラップさせることにより実現できる。更に、このコピー領域をハードウェアのバッファ上にとればスタックの先頭1フレーム分をキャッシングしていることになり、呼び出し元引数のコピー処理、およびユニフィケーションにおける引数へのアクセスを高速化できる。PSIではこの発想に基づきスタックの先頭1フレームを保持する特殊なハードウェアバッファ（フレームバッファ）を導入した。また、このようにユニフィケーションに先立って呼び出し元の引数をコピーする方式を"引数コピー方式"と呼ぶことにした。この方式の利点は呼び出し側引数の評価が1度で良いこと、ハードウェアサポートが期待できることである。

これを更に進め、従来型汎用レジスタマシンを念頭において、呼び出し側引数を普通のレジスタ（引数レジスタと呼ばれる）にロードしてしまう方式をDavid H.D. Warrenが提案した。レジスタの最適な利用の仕方をすべてコンパイラが決定するところが特徴である。この方式では普通のレジスタファイルを用いれば良いためハードウェアを単純化できること、レジスタの使用を自由に決められるため、例えばユニフィケーションの間に次の述語呼び出しのための引数を準備するなどの最適化が行えることである。汎用機でも効率の良いProlog処理系を実現できることやVLSI化に適したアーキテクチャとなることからこの引数レジスタ方式が主流になりつつある。

6．キャッシュメモリ、スタック、引数レジスタの比較

Prologマシンの実行速度をあげるためにはスタック管理の効率化、アクセスの高速化が重要となる。そのためには、キャッシュメモリを装備する、専用のハードウェアスタックを設ける、引数レジスタを設けるという3つのサポート方式が考えられる。以下に、各方式の特徴を要約する。

1）キャッシュメモリ

キャッシュメモリの利点は、すべてのメモリアクセスを均等に高速化できることである。これまでの汎用計算機に於いては既にその効果が確立しており、CPUの実行速度とメモリアクセスタイムの時間差を吸収するのに役立っている。Prologマシンにおいてはスタックアクセスの局所性が未知数であり、また述語の呼び出し処理はこれまでのプログラミング言語における手続き呼び出しの連続に相当するため命令コードへのアクセスが分散しがちである。しかしながら、PSIにおける評価の結果、メモリアクセスの局所性は従来型言語の場合と同様にかなり高いことが分かりキャッシュメモリの有効性が実証された。キャッシュのヒット率は予想通りローカルスタック、グローバルスタック、ヒープの順で高く、命令コードアクセスの局所性が低い。

2）ハードウェアスタック

既に述べたように従来型言語における関数もしくは手続き呼出しをサポートするような単純なスタック機構では不十分であり、Prologの実行を考慮した特殊なスタック機構が要求される。PSIではスタック先頭1フレーム分のみをバッファリングするハードウェアバッファ（フレームバッファ）を設けた。ユニフィケーションに先立って呼出しに必要な引数はフレームバッファにロードされてからアクセスされる。この結果、例えば引数の多い述語のユニフィケーションの高速化、ユニフィケーションの失敗によるバックトラック（shallow backtrack)の高速化、述語の再帰呼出しの高速化（tail recursion optimization)に効果がある。但し、PSIの場合にはフレームバッファの切り換え等のハードウェアサポートを余り持っていないシンプルな構造のためフレームバッファの利用に若干のオーバヘッドが生じ、期待したほどの十分な効果は得られていない。結果として、このようなハードウェアサポートは次に述べる単純なレジスタ構成に比べ必ずしも最適な実行手段とはならないが、それなりの効果と、素直な実現手法によるソフトウェア側の負担の軽減がはかれるのが大きな魅力である。

3）引数レジスタ

呼出しに必要な引数データをスタックに直接格納せずに一旦レジスタ上に置き,これを機械語命令により直接操作する方式をとると、最適化が十分に利く場合、非常に効率の良い実行が達成できる。ハードウェアを設計する立場からもメカニズムの単純化によるハードウェアの高速化、ハードウェア量の軽減がはかれる。但し、レジスタを効率良く利用しようとすると同一レジスタを破壊使用することが普通になるためバックトラックの高速化には必ずしも適していない。また、ガベージコレクションを行う立場からはどのレジスタが使用中であるかの識別に工夫が必要となる。引数レジスタ方式は高速であるがレジスタの管理に注意が必要である。

ハードウェアスタックも引数レジスタ方式も共に、ルール右辺での次の述語呼出しが行われる場合には、結局、その内容をスタックに保存することが必要となる。この観点からはむしろ普通のスタックを高速にアクセスするハードウェアサポートを検討することのほうがPrologやLispのようなスタックを多用するプログラミング言語の高速実行には重要であろう。

7．アーキテクチャサポートの実例

これまでの節でPrologの実行に必要なメカニズムと、その高速な実行にはタグを設けること、スタックの高速なアクセス等が重要であることについて述べた。本節では実際に開発されたマシンで具体的にどのようなアーキテクチャサポート

が行われているかについて幾つかの例をとりあげる。

1) タグアーキテクチャ

PSIではメモリの1語を32bitsのデータ部と8bitsのタグ部からなる40bits構成とした。タグ部は専用のハードウェアにより高速にデータタイプ判定が行えるようになっている。例えば、次のようなマイクロ命令により1マイクロ命令サイクル内で演算と並列にデータタイプの判定（この例ではintegerをチェックしている）が可能である。

```
cDR := LTR + GR04,
     if (tag(GR04) == INT) goto interger_type.
```

またいくつかのデータタイプに対応して処理が異なるような場合には同様に1マイクロ命令サイクルでデータタイプに応じた多方向分岐を実現できる。

```
GR00 := int_3 + GR01,
     case (tag(cDR.tbl3)) base case_table.

:: begin_case #16 step 2.
     case_table:
          ----> INTEGER  : .....
          ----> ATOM     : .....
          ----> UNDEFINED: .....

:: end_case.
```

ユニフィケーションはこのようなマイクロ命令の分岐テーブルにより実現されている。

また、CHIではこれを更にすすめて、2つのデータのタグビットパターンの組合せによる多方向分岐機能やデータ部の値の比較も同時に行う多方向分岐機能などもサポートしている。

データタイプチェックはPrologマシンでの基本操作であり上記のようなハードウェアサポートによりすべてのタグ操作が1マイクロ命令サイクルで完了する。

2) メモリアクセスおよびスタックサポート

PSIではPrologのスタックをサポートするために専用ハードウェアバッファを備えている。これはスタック先頭の1フレーム分をキャッシングするためのバッファであり、PSIはこれを2面装備しており、呼出し元環境がまだバッファ上に存在する場合にはバッファからバッファへ必要な引数データをコピーすることが可能である。

PEKではハードウェア化の効果を検証するためにProlog処理に必要なスタックを全てハードウェア化している。またユニフィケーションにより代入が生じた変数セルアドレスをトレイルスタックへ格納する操作もハードウェア化により自動化している。

スタック管理のために必要となる各種のベースレジスタ群は一種のキャッシングでありコントロールスタックの1フレーム分を装備していると考えても良い。CHIではこれらのベースレジスタ群の切り換えを高速にするため2組のベースレジスタファイルを装備している。

述語の呼出しやユニフィケーションの実行のためには呼出し元と呼ばれた側の2つの環境をアクセスする必要がある。このためにPSIでは2組のメモリインタフェースレジスタ（メモリデータレジスタ、メモリアドレスレジスタ）を装備している。これはCHIやPEKでも同じである。

引数レジスタ方式を採用しているのはPLMやCHIであるが、これらは2ポートレジスタファイルを用いて実現されており、レジスタファイルそのものは特別の機能を有していないが出力データのデータタイプ判定を高速化するためのパスを設けるなど細かい点に高速化の工夫がなされている。

3) 処理の並列化

ハードウェア構成による高速化の工夫としてCHIではデータ処理部とスタック管理のためのアドレス計算部とを分離し並列動作を可能としている。また、PLMではマシンサイクル短縮のために全体を3ステージのパイプライン構成としTTL素子を使いながらマシンサイクル100nsecを達成している。

4) マイクロプログラム技術

ユニフィケーション処理ではデータタイプの組合せに応じて種々の異なる処理を要求されるが、その各々の処理は単純である。この様な分岐が多く浅い処理にはマイクロ命令の強力な分岐機能が有効である。

マイクロ命令の持つ並列性、柔軟性、および強力な分岐機能はユニフィケーションのみならず、Prologの処理全般に渡ってその高速化に有効である。例えば、PSIではメモリの動的割り付け処理やガベージコレクタをマイクロプログラムで実現している。

8．専用マシン化の意義

Prologマシン、特に逐次型Prologマシンと従来型汎用マシンとのアーキテクチャの違いはそれほど大きくない。コンパイラが最適化を十分に引出す引数レジスタ方式の場合にはますますこの差は狭まり、専用のアーキテクチャサポートがな

いマシンでも比較的高速のProlog処理系を実現することができる。但し、そのためにはメモリアクセスの高速化や大容量化は必須となる。一般に汎用マシンでは大きなユーザメモリ空間を提供するために仮想記憶管理システムを採用しており、その結果レジスタ間データ転送は非常に高速であるがメモリアクセスはそれほど高速ではないことが多い。また動的なメモリ管理は従来型プログラミング言語においてあまり要求されないためPrologの処理に必要な多数のスタックを効率良く実現するためのアーキテクチャを備えていないことが多い。

いずれにしてもアーキテクチャによってスピードアップがはかれるのは高々２倍位でありデバイス技術、マシンサイクルによってこの差は埋められる可能性がある。即ち、鍵になるのは結局、CPUのサイクルタイムとメモリアクセスタイムとなる。例えば、汎用超大型機は最新のデバイス技術と高度なパイプライン技術を駆使しているためPrologにつてもコンパイラにより十分に最適化されたオブジェクトコードなら高速に実行することができる。[9] 一方、PSIのような専用機を最先端のデバイス技術、実装技術を利用して開発することは難しいのが現状である。

このような観点から、専用マシンの狙いは絶対スピードよりも高いコスト／パフォーマンスとパーソナルマシンとしての使い易さにあると言える。高価で高速な大型機は通常、複数のユーザ間で共有されるためマシンとしての単体性能は確かに高いが実際の利用環境においてユーザはそのパワーを十分利用できない。これに対して専用のパーソナルマシンはその計算パワーをフルにひとりで占有することが可能となる。

使い易さの観点からも専用機を占有する方が応答性が良く、開発の能率が上がる。特に良好な対話型プログラミング環境を提供するためにはマンマシンインタフェースに多くのCPUパワーを使用することが必要であり、この意味からもパーソナル型の専用機として構築することが望ましい。強力なエラーチェック機構やデバッグサポート機構も専用マシンでは低コストで実現できるのが魅力である。

９．今後の展望

1）高級言語マシンの限界

Prologマシンは対象としているProlog言語を効率良く実行することを最大の目標に設計される。例えば、Prologを用いてより高度な応用システムを構築した場合にその応用問題に依存した基本機能が抽出できたとしても、それを直接サポートすることはできない。これは高級言語マシンの限界であって、Prologマシンは対象としているPrologの言語仕様を越えることができない。（勿論、ある単機能を組込み述語として直接マイクロプログラムでサポートすることは可能である。）

またマシンの設計思想は最終的にPrologをどう使っていくかに強く依存することになる。例えば完成したプログラムを高速に実行するようなコンパイルを主体とした利用環境か、あるいはプログラム開発を主体としたインタプリティブな利用環境かでPrologマシンの特性は異なるであろう。

プログラミング言語はあくまでもツールであって、Prologを用いたからといって問題が解決するわけではない。しかしながら、言語は思考のためのツールであり、対象としている問題が素直に記述できるということはそれだけソフトウェア開発者の負担を軽減し、開発／保守の能率を向上させることにつながる。この意味でより使い易く強力な論理型言語の開発とそれをサポートする専用マシンの開発が要求されている。

2）RISC + タグアーキテクチャ

Prologマシンをとりまく世界の研究開発動向は現在RISC指向のアーキテクチャに関心が集っている。一つの理由はVLSIの利用を進めるためには複雑なマシンアーキテクチャがまだ実現困難であることによる。もうひとつの理由はコンパイラ技術の向上により単純なアーキテクチャであっても高い性能が達成できることが可能になったことにある。確かに商用ベースのマシンであれば単純でハードウェア量の少ないマシンのほうが複雑な専用機よりも有利である。

しかしながら、これまでも単純で汎用のハードウェアを使いながら、ソフトウェアの負担によって計算機の利用範囲を拡大してきた。その結果としてソフトウェアの低い生産性が計算機のより高度の利用を阻むようになってきており、ハードウェアとソフトウェアのセマンティックギャップが問題となってきた。RISC指向のマシンアーキテクチャはこのギャップを縮めることにはならない。

論理型プログラミングという新しいプログラミングパラダイムに注目したのはユニフィケーションとバックトラックに代表されるPrologの持つ動的な特徴ではなかったか。これからのマシンアーキテクチャを考える上で重要なのは何が最も基本的な処理単位かという点である。より高度なシステムを構築するためにはベースとなる基本単位もより高いレベルにならなければなるまい。少なくともPrologの持つ推論メカニズム、あるいはユニフィケーションメカニズムがそのような基本処理の一つであることは確かであろう。これらのメカニズムを基本部品として自由に活用できるようにすることが、まさに、Prologマシンの使命といえる。

表１．逐次型Prologマシン

	開発	素子	主記憶	WCS	CycleTime	nreverse	（処理方式）
PSI	ICOT	TTL	40bits*16M	64bits*16K	200nsec	33KLIPS	μ-interpreter
CHI	ICOT	CML	36bits*64M	80bits*11K	100nsec	200KLIPS	Warren-code
PEK	神戸大	TTL	20bits*32K	96bits*16K	170nsec	60KLIPS	μ-interpreter
PLM	UCB	TTL	32bits*??	144bits*1K	100nsec	240KLIPS?	Warren-code

参考文献

1) S.Uchida, M.Yokota, A.Yamamoto, K.Taki and H.Nishikawa : Outline of the Personal Sequential Inference Machine : PSI, New Generation Computing, 1-1, 1 (1983)
2) M.Yokota, A.Yamamoto, K.Taki, H.Nishikawa and S.Uchida : The Design and Implementaion of a Personal Sequential Inference Machine : PSI, New Generation Computing, 1-2, (1983)
3) D.H.D.Warren : An Abstract Prolog Instruction Set, TR-309, AI Center, SRI International (1983)
4) T.P.Dobly, A.M.Despain and Y.N.Patt : Performance Studies of a Prolog Machine Architecture, Proc. of 12th Annual Symposium on Computer Architecture (1985)
5) R.Nakazaki, A.Konagaya, S.Habata, H.Simazu, M.Umemura, M.Yamamoto, M.Yokota and T.Chikayama : Design of a High-speed Prolog Machine (HPM), Proc. of the 12th Annual Symposium on Computer Architecture (1985)
6) Y.Kaneda, N.Tamura, K.Wada and H.Matsuda : Sequential Prolog Machine PEK Architecture and Software System, Proc. of the International Work-Shop on High-Level Computer Architecture '84 (1984)
7) 横田、内田　：推論マシン、計測と制御、25-4 (1985)
8) H.Komatsu, N.Tamura, Y.Asakawa and T.Kurokawa : An Optimizing Prolog Compiler, Proc. of the Logic Programming Conference '86 (1986)