計算機アーキテクチャ 77－11
（１９８９．７．１４）

Ｐｅｇａｓｕｓ　Ｐｒｏｌｏｇ　プロセッサ

――― ＶＭＥｂｕｓボードによる評価 ―――

横田 隆史，　瀬尾 和男
三菱電機（株）中央研究所

本稿では、ＲＩＳＣ手法の導入によりＰｒｏｌｏｇの比較的単純で定形的な処理の高速実行を実現したＰｅｇａｓｕｓにおいて、ＲＩＳＣの制御構造の単純さ故に、デリファレンスやトレール処理など頻度の高い処理で無効命令が発生し性能上の障害となりやすいことを示し、この解消のため新たに採用された動的命令差替えやスカッシング等の方法を示す。これに伴う回路上の変更は小規模に抑えられ、改善されたＰｅｇａｓｕｓチップはＶＭＥｂｕｓボードに実装されシステム化されている。このシステム上で評価を行った結果、相当な改善効果が認められるとともに、システムとして満足すべき推論性能が確認された。

Pegasus Prolog Processor
— Performance Evaluation on VMEbus Board —

Takashi Yokota and Kazuo Seo
Central Research Laboratory, Mitsubishi Electric Corp.
1-1, Tsukaguchi-Honmachi 8-Chome, Amagasaki, Hyogo 661, Japan.

Pegasus is a Prolog processor based on RISC architecture. RISC mechanism offers both fast execution and small implementation, whereas it causes inefficiency in frequent Prolog-oriented operations, such as dereferencing and trailing. In this paper, to overcome the inefficiency, we introduce dynamic instruction modification for dereferencing and branch control for trailing. These mechanisms need slight modification on Pegasus, and new Pegasus chip is fabricated.

Using the new chip, Pegasus inference engine is built as a VMEbus board. A performance evaluation on this system shows that the new mechanisms work efficiently and make a considerable contribution on the system performance.

1．はじめに

Pegasus Prologプロセッサ[1~3]は、Prolog コンパイル後の WAM (Warren Abstruct Machine [4])コードを効率よく実行するようにアーキテクチャ設計されている１チップ・マイクロプロセッサである。RISC (Reduced Instruction Set Computer)手法によってアーキテクチャ設計することによって Prolog の高速な処理機能を１チップというコンパクトさで実現している。

Pegasusでは、WAM モデルによる Prolog 処理のメカニズムを解析し、これに最適なアーキテクチャ/命令セットを RISC の枠内で実現可能な形で採用するアプローチによって高速化が行われている。Pegasus は RISC 手法の採用によって高性能とチップ規模の縮小化を目指したものであるが、その反面 RISC 手法により徹底して制御を簡素化した故に生じるオーバーヘッドが無視できなくなってきた。特に分岐動作に伴うペナルティスロットにおいて、ディレイドジャンプ手法による最適化が適応できないケースが比較的多く発生する。

そこで、RISC の簡素な制御構造とデータパスを活かしつつこの問題点を解決するため、動的命令差替え機構による命令レベルの強化[3]と、スカッシング[7]による分岐ペナルティの削減・有効利用を図った。

Pegasus チップは、上の問題点を回路の小規模な変更・追加により解決し、さらにハードウェア割込みを新たに加え、1.5μルールCMOSで設計・試作されている。チップはトランジスタ数約８０K、サイズ9.7㎜×9.8㎜で、マシンサイクル200nSでの動作が確認されている。

このチップをバックエンド推論プロセッサとして用いるための推論ボードもすでに試作されており、システムとしての開発環境が整備されてきている。

本稿では、性能向上のための改善点について述べ、さらに VMEbus を用いた評価用の推論ボードと、Pegasus での Prolog 処理システムについて触れる。そして他のシステムとの比較のため、Warren のベンチマークプログラム[5]について、ボード上で動作させた際の性能評価について報告する。

2．ＲＩＳＣオーバーヘッドとその解決

RISC では基本的に１命令１サイクルが原則であり、複数サイクルを要する処理についてはいくつかの命令の組合せによって実現することになっている。実際、これまで RISC が扱ってきた範囲では、アーキテクチャ上のサポートと、高速な命令の組合せ、さらに最適化を行うことによって、多くの場合満足すべき性能を得ることができた。

Pegasus のようにある程度応用を特定したプロセッサにおいては、処理対象への障害をできるだけ排除する必要がある。特に Prolog 処理においては、比較的定形的な処理パターンが多く出現し、この中には出現頻度が多くピュアな RISC 命令†の組合せだけでは一種のオーバーヘッドとなるものが存在する。このような処理パターンについて、Pegasus では RISC の簡素で高速な実行環境を維持するため、ハードウェアの小規模な追加・変更による対策が考慮されている。

2.1. デリファレンス

Prolog の処理では変数が別の変数を指し、その変数がまた別の変数を指す、といったポインタ連鎖の形成が許される。このため、「何か」がポインタ連鎖によってバインドされている変数に対して何らかの処理を行う場合、連鎖をたどって最終的に指している実体を得る必要がある。

このデリファレンス処理は、コンパイル時のオプティマイズによってある程度削減できるとはいえ、非常に出現頻度が高い。Pegasus システムにおいてこの処理は、

「あるレジスタが参照タグ(#ref_tag)を持っていれば、そのデータ部の値をアドレスとするメモリ番地からデータを読出し、同じレジスタに書込む」

という処理を最初のタグ条件が成立する限り繰返すこ

---

† 本稿ではすべての命令が同一のマシンサイクルで終了し、処理内容の変更が分岐命令のみに許されている形式の RISC をピュア RISC と呼ぶことにする。

とになる。これをピュア RISC 式にコード化すると

```
L1: Branch  if(Reg.tag ≠ #ref_tag), L2  ①
    Branch  if(Reg.tag = #ref_tag), L1  ②
    Load    Reg ← (Reg)                 ③
L2:
```

という形になる。この例で条件の異なる分岐命令が重複しているのは、命令のパイプライン実行により分岐に伴うペナルティスロットが1命令分あり、2行目の命令は最初の分岐の条件成立・不成立に拘わらず常に実行されるためである。デリファレンスがループする場合、②が成立し L1 に戻るがその際のペナルティスロット③によってレジスタを更新する。結局 n レベルの参照連鎖に 3n+2 命令分の時間を要することになる。

このループを1命令の実行制御内に組込めれば所要時間が削減できるほか、コード量の削減も期待できる。Pegasusではデータパス上にデリファレンス専用のタグ比較回路を設け、比較結果を制御部分に戻すことにより、デリファレンス命令ひとつでデリファレンスループを実現している。これを用いることにより、上に示したコードは

```
deref  Reg, #tag_ref                ①
```

に置換えられる。デリファレンス命令①は、まずレジスタ Reg を読出しタグを tag_ref と比較する。以降の命令実行はこの比較結果に依存し、不一致ならばそれ以降何もせずに次の命令に移るが、一致した場合 Reg のデータ部をアドレスとしたメモリ読出しを行い、読込まれたデータを元の Reg に書込むとともに同じ動作を繰返す。

デリファレンス命令で行われる1命令内ループは、命令の表現上、すべての命令が同一の実行時間で終了するという RISC の基本概念ともいえる特徴を超えているように見える。しかし実際には、動的命令差替え[3]により命令実行制御のワイヤ論理を利用して命令パイプライン内の実行内容の変更を行い、またパイプラインのオーバーラップをうまく利用することによってパイプ間にわたる動的命令差替えを適応し、1命令内ループを実現している。

Pegasus の命令実行は図1のように6ステージに分割されて2ステージ毎に命令を投入する形でパイプライン実行される。図中F(Fetch),D(Decode)ステージで命令のフェッチ/デコードを行い、R(register Read)ステージでレジスタファイルの読出しとメモリアドレスの計算が行われる。演算やメモリアクセスは次のA(ALU/memory)ステージで行われ、調整用のN(Nop)ステージを経たのちW(register Write)ステージでレジスタファイルに書込まれる。前の命令のWステージが次の命令のRステージの後に実行されることにより、レジスタ参照の際にインターロックが生じる恐れがあるが、既に確定した書込みデータをレジスタへの書込み前に参照するためのバイパス回路を設けることによりインターロック・フリーにしている。

デリファレンス命令では、R_1ステージでレジスタを読出すと同時に比較が行われる。この結果に従ってD_1ステージで生成された命令制御コードを単純な論理演算で変更することにより、続くA_1,W_1ステージでの動作を決定する。

R_1ステージでの比較結果は次のステージが開始するまでに次の命令スロットのD_2ステージに伝えられる。デリファレンスループが続く場合、F_2で読込んだ命令を捨て、もう一度同じ命令(デリファレンス命令)をデコードする。この間プログラムカウンタはF_2のアドレスで凍結される。

こうしてループが続く限り次のアドレスに対して命令のダミーフェッチを行いながら参照連鎖を解いてゆく。

デリファレンスループがR_2ステージで解かれたなら、F_3ステージでフェッチされた命令は捨てられることなくD_3ステージでデコードされる。プログラムカウンタの凍結もこの時点で解かれる。

結局、参照のレベルnに対して要する時間は上に示した3n+2からn+1に削減される。

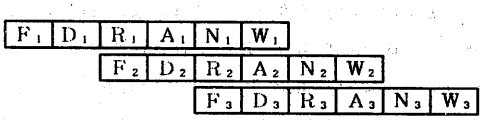

図1．Pegasusの命令パイプライン

2.2. ジャンプ・ペナルティスロットの有効利用

RISC の多くは命令のパイプライン実行により、分岐命令の次の命令(ペナルティスロット)も実行される。単方向分岐ならば、ペナルティスロットを積極的に使うディレイドジャンプ手法によって、かなり有効に活用できるが、複数の分岐条件を扱う場合、ペナルティスロットを有効な命令で埋めることができず、NOP のまま残るケースが多く発生しやすい。

こうした分岐は Prolog においてアンバウンド変数に値をバインドする際に行われるトレール判定において顕著に現れる。Pegasus において、アンバウンド変数はヒープ、ローカルスタックのいずれかの領域に確保される。そこでピュア RISC によるコードは2つの領域の置かれるアドレスの大小を利用して、

```
        Branch   if(Reg＞Htop), L1        ①
        NOP                               ②
        Branch   if(Reg≧Hbak), L2         ③
        NOP                               ④
L1:     Branch   if(Reg≧Lbak), L2         ⑤
        NOP                               ⑥
        Store    Reg ⇨ (TR++)             ⑦
L2:
```

となる。ここで Reg はトレール対象となるレジスタ、Htop/Hbak はヒープのトップ/バックトラックポイント、Lbak はローカルスタックのバックトラックポイント、TR はトレールスタック・ポインタ(いずれもレジスタ)である。

上の例では、①で変数が置かれている領域を判定し、③でヒープ変数についての、⑤でローカルスタック変数についてのトレール判定を行っている。

このような処理の場合、ディレイドジャンプ手法による最適化で②④⑥の NOP 命令を他の命令に置換えることはほとんど期待できない。そこで分岐のペナルティスロットにある命令の制御を行うスカッシング[7]の考えを採用した。スカッシングは分岐命令において分岐条件が成立した場合、次に実行されるペナルティスロットを強制的に NOP 化するものである。ここで、無条件分岐は常に条件が成立するものとして扱う。

Pegasusでは分岐命令が連続する場合にのみスカッシングを適応可能としており、これにより上の例は、

```
        Branch/sq  if(Reg＞Htop), L1       ①
        Branch/sq  if(Reg≧Hbak), L2        ②
L1:     Branch     if(Reg≧Lbak), L2        ③
        NOP                                ④
        Store      Reg ⇨ (TR++)            ⑤
L2:
```

と記述される。①②はスカッシング付き分岐命令である。例えば命令①で条件成立し分岐が行われると次の命令②での条件判定を無効化し、実質的に何もしない NOP 命令と等価にする。命令①で条件成立しなければ命令②は①からの介入なく所定の条件判定→分岐動作を行う。

さらに Pegasus では条件判定に2つのレジスタのタグ部が等しいことを AND 条件として加えることも可能としている。つまり2つのレジスタが同一のタグを持ち、かつ、データ部が与えられた条件を満たせば条件成立し分岐する。結局トレール処理は、

```
        Branch/sq/teq  if(Reg≧Hbak), L2    ①
        Branch/teq     if(Reg≧Lbak), L2    ②
        NOP                                ③
        Store          Reg ⇨ (TR++)        ④
L2:
```

となる。①はタグ一致条件とスカッシングの両方を備えた分岐命令である。

結局、命令数で7から4に、処理時間で1～3命令分の削減となっている。

スカッシングは分岐命令のみに制限することにより容易に実現できる。すなわち、命令の条件判定結果を一時的に保存しておき、次の命令の条件判定に用いるようにすれば良い。

タグ一致条件については、Pegasus がタグ/データの各々に独立した ALU を持っていることを利用している。

2つの ALU はほぼ同等の演算機能と条件判定機能を持っており、タグ/データで異った条件の判定を並列に行い両者の積をとることも、さほどの回路負担にならずに実現できる。

3．Ｐｅｇａｓｕｓシステムの実装

3.1．ＶＭＥｂｕｓボード

Pegasus チップの評価とこれを用いての高速推論システム開発のため、VMEbus 仕様のバックエンド型推論ボードが試作されている。VMEbus はすでに unix-EWS の標準的外部バスとなっており、システムのマシン依存性を最小限に抑えることができる利点がある。さらにボードはバスマスタ機能を持たない完全な VME bus スレーブとなっており、ダブルユーロカード１枚のコンパクトさで Pegasus の実行環境を実現している。

ボードは図２に示すように、Prolog 処理の核となる Pegasus CPU と 256KW RAM、それに演算機能の強化のために浮動小数点演算プロセッサ(FPC)を加えた、比較的簡潔な構成となっている。

ボード上のバスインターフェースは、ホスト計算機から Pegasus を制御・モニタするためのレジスタを持つ。ホスト計算機は、この制御レジスタに適当な制御ワードを書込むことによってチップのリセットや起動/停止などの制御を行い、状態レジスタを読出すことによって Pegasus の動作状態(動作中/停止状態、あるいはホスト計算機からの割込みを受付けたか、等の情報)を知ることができる。

またバスインターフェースは、RAM をアクセスするための機能を持っている。ただし、Pegasus とのアクセス競合を防ぐため、Pegasus が停止状態にありメモリアクセスを必要としない時にのみ VMEbus 側(ホスト計算機)からのアクセスを許可する。なお本システムでは Pegasus の動作中にホスト計算機が RAM 内の情報をアクセスする必要は生じない。

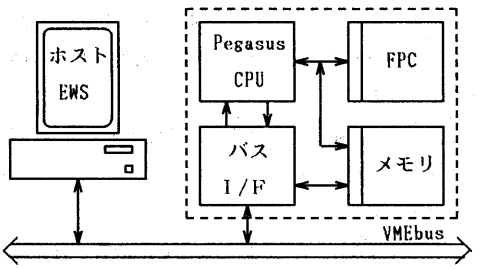

図２．ボードの構成

ホスト計算機はこれらの機能を用いて以下のように Pegasus を制御し、Prolog 処理を実現している。

① Pegasus, FPC を含めたボードのリセット。
　制御レジスタの初期化。
② Pegasus コードの RAM 上へのダウンロード。
③ Pegasus の起動。
　Pegasus 停止まで待つ。
④ RAM アクセスにより停止原因を知る。
　Pegasus からの処理要求がある場合、要求コードに従った処理を行い③に戻る。
　実行終了ならば結果を表示して終了。

このほか、ボード上には Prolog のスタック境界を検出しガーベージコレクションのトリガとするためのスタック境界チェッカや、プログラムのバグや暴走の検出を主眼とした不正アドレスチェッカ等のメモリアクセス・モニタ機構、Pegasus の状態変化によって VMEbus 経由でホスト計算機に割込みをかけるインタラプタの機構、さらに、評価用として Pegasus の動作時間の計測用のタイマ機構が設けられている。

3.2．ソフトウェア環境

Pegasus の命令セットは WAM コードの高速実行を目標に設定されており、従って性能を十分に発揮させるためにはコンパイルコードの使用が前提となる。従って、ここではコンパイルコードによる実行について述べ、デバッグ用に別途開発されているインタープリタ環境については割愛する。

Pegasus における Prolog コンパイルは、①prolog ソース → ②WAM 中間コード → ③Pegasus 命令の手順を踏む（図３）。コンパイラは Prolog ソースファイルとともに WAM → Pegasus のマクロ展開テーブルを読込み、一旦WAMコードを生成したのち展開テーブルに従って機械的に Pegasus アセンブラコードを出力する。そしてこのコードを組込み述語やスタートアップモジュール等とリンクすることにより最終的な Pegasus の実行オブジェクトとシンボル情報が得られる。

前節に示したように、Pegasus はボード上で自律的に動作できないため、ホスト計算機側で Pegasus オブジェクトコードのダウンロードや実行の起動/停止等の制御を行う必要がある。またホスト計算機は、人

出力を持たないボードのために、外部組込み述語の形で Pegasus から要求される入出力のサービスやマンマシンターフェースの役割も担っている。

結果的に Pegasus における Prolog 処理は、ボード上にダウンロードされた Pegasus コードと、これを制御/機能サポートするためのモニタプログラム(ホスト計算機上)とのインタラクションにより進められる。この概略のソフトウェア構成を図４に示す。ホスト計算機－Pegasus間の交信は、双方に配置されるドライバと呼ばれる部分の間で、定められた規約に従って行われる。

ホスト計算機上のモニタは、起動時に指定された Pegasus オブジェクトコードをボード上の RAM にダウンロードし、シンボル/アトム情報を読込む。その後は一般の Prolog システムと同様にターミナルにプロンプトを出してユーザからの問合せ入力を待つ。この間 Pegasus は停止している。

モニタは入力された問合せを解釈し、Pegasus の実行のための環境を整えると同時に Pegasus を起動し、この設定内容を Pegasus 側に伝える。Pegasus はホスト計算機から伝えられた内容に従った初期設定を完了すると一旦停止し、ホスト計算機からの実行開始指示を待つ。ここでモニタが Pegasus を再起動すると Pegasus 上で問合せに従った Prolog 実行が行われる。

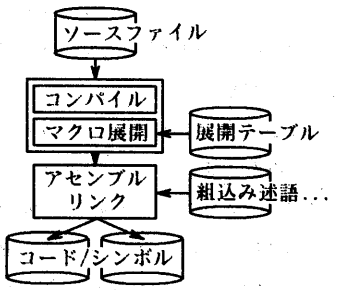

図３．コンパイル過程

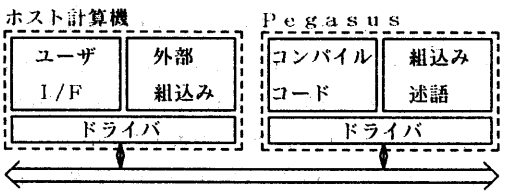

図４．動作環境

Pegasus は実行を終了するか、あるいは外部組込み述語をホスト計算機側に要求する場合停止し、モニタに何らかのアクションを求める。通常モニタは Pegasus が実行を開始した後、自ら停止するまで待つ。

モニタは Pegasus の停止を確認したら RAM の特定番地に書込まれている停止コードを読出し、停止原因を知る。処理の終了ならば結果を RAM 上から読出し表示した後、もとの入力待ちの状態に戻る。外部組込みの実行要求であれば停止コードに従った処理を行い、その結果を所定の RAM エリアに書込んだ後 Pegasus を再起動する。

４．性能評価

4.1. 性能測定環境

上に示した VMEbus ボード内には、性能測定のためのタイマが組込まれている。このタイマは Pegasus の動作時間を１μs 刻みで計測できる。ホスト計算機上で Pegasus を制御するモニタ(図４)を若干変更し、Pegasus 起動前にタイマをクリアし、Pegasus が処理を終え自ら停止するまでの時間を表示するようにした。これにより、Pegasus が動作した正味の CPU タイムを正確に測定できる。評価プログラムに対する Pegasus コードはコンパイラが生成したものをそのまま用いている。

整数間の加減算や比較はPegasus自身で行い、実数を含む演算や乗除算はFPCを用い結果を実数で得ている。

4.2. ベンチマーク

ベンチマークプログラムは他のシステムとの比較のため[5]に示されているものを用い、以上のようなハードウェア/ソフトウェア環境においてソースプログラムからコンパイルし、ボード上で動作させ実行時間を測定した。

２.に示した命令上の改善点が性能上どの程度反映されるか調べるために、デリファレンス、トレールの処理に限って

ｉ．デリファレンス、トレールともに改善されたコードを用いる場合

ii．デリファレンスに改善前のコードを用いる場合
iii．トレール処理に改善前のコードを用いる場合
の３通りの方法でコンパイルした。

これらの実行時間測定結果を他のシステムのデータと併せて表１に示す。

表からわかるように Pegasus での Prolog 実行は、プログラムの実行時間の多くがユニフィケーション/バックトラックといった Prolog の基本的な処理構造に費やされる場合、非常に高い性能を達成できる。実際、マシンサクルが倍の PLM 等のマシンと比較しても同等あるいはそれ以上の速度が得られている。

しかし Prolog の処理メカニズム以外の要素が速度上大きな比重を占める場合、必ずしも期待されるほどの効果が得られていない。今回行ったベンチマークの中では dbquery がこれにあたる。すなわち、このプログラムは乗除算と計算結果に対する数値比較が処理の中で大きなウェイトを占めている。次の項でこれについての検討を行う。

4.3. 数値演算についてのケーススタディ

dbquery では、①データベースから値(整数)を得る、②乗算･除算、③比較･条件判定、④成立すれば更に乗算(２回)、⑤比較･条件判定、という処理が行われる。

上のベンチマークでは、乗除算はデータのタイプ(整数/実数)によらず FPC 上で演算を行い、結果を常に実数で持つことにした。このため、①で得られた整数値も②以降の処理ではすべて FPC 上で実数演算されることになる。

FPC を用いての実数演算は、①第１オペランドを FPC 内のレジスタに送る、②第２オペランドを送ると同時に演算を開始する、③演算結果を FPC から受取る、という３ステップの手続きによって行われる。乗算の場合ではこれらの処理に約 16μs 要しているが、その約14%が FPC－Pegasus のインターフェースによるオーバーヘッドになっている。転送を含めた FPC 演算時間は dbquery 処理全体の実行時間の 55～60% を占めている。

この方法では１回の演算毎に生じるデータ転送と、やりとりされるデータのフォーマット変換(FPC 内部で行われる)がそのままオーバーヘッドになっている。このオーバーヘッドは計算上60%にも及び(乗算)、演算自体よりも転送に時間が費やされていることになる。

そこで、FPC へのデータ転送をオプティマイズしたコードを用いて dbquery を実行した。また比較のため、FPC を使わず整数演算のみを用いた場合の性能も測定した。結果を表２に示す。

転送のオプティマイズにより約16%の性能向上が認められた。すなわち演算時間の25～30%が削減されことになる。

	ｺﾝﾊﾟｲﾙｺｰﾄﾞ	転送ｵﾌﾟﾃｨﾏｲｽﾞ	整数演算
dbquery	69.46	58.65	49.51

表２．ＦＰＣの転送オプティマイズの効果

ベンチマーク	DEC-10 ｺﾝﾊﾟｲﾙ	Berkeley PLM	Pegasus ｺﾝﾊﾟｲﾙ ｺｰﾄﾞ	Pegasus 性能向上 ﾃﾞﾘﾌｧﾚﾝｽ	Pegasus 性能向上 ﾄﾚｰﾙ判定
nreverse	53.7	2.66	2.29	0.40 (17.5%)	0.18 (7.9%)
qsort	75.0	5.50	3.65	0.53 (14.5%)	0.25 (6.8%)
deriv					
→ times10	3.00	0.329	0.278	0.021(7.5%)	0.006(2.2%)
→ divide10	2.94	0.380	0.309	0.021(6.8%)	0.006(1.9%)
→ log10	1.92	0.109	0.143	0.010(7.0%)	0.003(2.1%)
→ ops8	2.24	0.213	0.186	0.015(8.1%)	0.005(3.1%)
serialise	40.2	3.19	2.46	0.34 (13.8%)	0.11 (4.5%)
dbquery	185	17.57	69.46	5.75 (8.3%)	1.80 (2.6%)

表１．ベンチマークテスト結果（単位:mS）

5．おわりに

Pegasus Prologチップの開発を通して、Prolog 処理の高速化のために RISC 手法を用いることの有効性を実証した。本稿では Prolog 処理においてピュアな RISC の制御形態を用いることにより、無視できないオーバーヘッドが多頻度で発生する恐れがあることを指摘し、RISC の利点を活かしつつ最小限の変更でオーバーヘッドの発生を防ぐため、Pegasus において採用された方法を紹介した。

このうちデリファレンス命令については Prolog 専用命令としての性格を持つが、スカッシングを始めとする分岐ペナルティの回避方法は直接 Prolog の処理に関係しない部分においても有効に用いることができ、相当の効果を上げられるものと考えられる。前節で示した評価の上では都合上トレール判定の部分のみピュアな RISC のコードとしたが、実際にはこれ以外にもスカッシングやタグの同時比較を用いている部分は多く、速度とコード量の削減に寄与している。

今後 Pegasus を改善して行く上で、評価の際にも問題となった ①演算能力の向上、パイプラインピッチがますます短くなることに対する ②外部アクセスのバンド幅の確保、の２点が大きな課題となっている。

特にメモリのアクセスに関しては、命令のフェッチ周期が非常に短い RISC 方式によって、メモリへのデータアクセスが非常に多い Prolog 処理を行うことにより、メモリのアクセス頻度は両者の和となり、ネックになりやすい。このためオンチップキャッシュの導入などによってメモリアクセスの負荷を軽減してゆくことが望まれる。

[謝辞]

本報告をまとめるにあたって、日頃ご指導いただいている三菱電機(株)中央研究所平山正治グループマネージャ、ならびにご討論いただいたシステム基礎研究部の諸兄に感謝いたします。

[参考文献]

[1] 瀬尾,横田「Prolog指向RISCプロセッサ:Pegasus」情報処理学会アーキテクチャ研究会資料63-5, 1986.

[2] 瀬尾,横田「Prolog指向RISCプロセッサ"Pegasus"－プロトタイプ試作とその評価－」情報処理学会アーキテクチャ研究会資料69-11, 1988.

[3] 瀬尾,横田「Prolog指向RISCプロセッサ Pegasus－動的命令差替えによるProlog処理の効率化－」情報処理学会コンピュータアーキテクチャシンポジウム, 1988.

[4] D.H.D.Warren: "An Abstract Prolog Instruction Set," Technical Note 309, ArtificialIntelligence Center, SRI International, 1983.

[5] D.H.D.Warren: "Applied Logic — Its Use and Implementation as Programming Tool," Technical Note 290, Artificial Intelligence Center, SRI International.

[6] E.Tick: "Prolog Memory-Reference Behavior, "Technical Report No.CSL-85-281, Computer Systems Laboratory, Stanford University, 1985.

[7] P.Chow: "MIPS-X Instruction Set and Programmer's Manual," Technical Report No.CSL-86-289, Computer Systems Laboratory, Stanford University, 1986.

[8] T.P.Dobry, et al.: "Performance Studies of a Prolog Machine Architecture," Proceedings of the 12th International Symposium on Computer Architecture, pp.180-190, 1985.