計算機アーキテクチャ　71－18
（1988. 7. 8）

PROLOGにおけるシャロウバックトラックの高速化方式

SHALLOW BACKTRACKING OPTIMIZATION FOR PROLOG PROGRAMS

阿部　重夫　　桐山　薫　　黒沢　憲一
Shigeo ABE　　Kaoru KIRIYAMA　　Ken-ichi KUROSAWA

日立製作所日立研究所
Hitachi Research Laboratory, Hitachi, Ltd.

あらまし　本論文では、WarrenのProlog命令セットをベースにしてシャロウバックトラックを高速化する最適化方式について述べる。主な最適化手法は以下のとおりである。
（1）ヒープのリセット、変数の開放、アーギュメントレジスタの回復の削除など、無駄なバックトラック情報の回復を削除する。（2）ユニフィケーションの失敗ができるだけ早くみつかるように、ユニフィケーションの順序付けを行なう。（3）プロセジャ内の共通のユニフィケーションを検出し、シャロウバックトラック時に削除する。
上記手法の有効性を内蔵型Prologプロセッサ IPPで検証し、quick-sortで２倍の性能向上（921 KLIPS）が得られることが確かめられた。

Abstract　This paper discusses new optimization techniques to speed up shallow backtracking within the framework of Warren's Prolog instruction set. The major speedup techniques are as follows: (1) eliminating unnecessary restoring of backtracking information; (2) ordering a unification sequence so that its failure is detected as early as possible; and (3) eliminating common unification during shallow backtracking.
The effectiveness of the techniques was tested for some benchmark programs using an Integrated Prolog Processor (IPP). For the *quick-sort* program, a speedup gain of 2.0 and an inference speed of 0.9 million logical inferences per second were obtained.

１．　はじめに

Prologは、ユニフィケーションとバックトラックで特徴づけられる推論機能を持ち、エキスパートシステムの開発に適した言語である。推論性能が低いという欠点も、WarrenのProlog命令セット[1]により大幅に改善されつつあり、この命令セットをベースにした専用マシン、あるいは汎用マシン上の処理系の開発が進められている。[2-9]

Prologの実行の高速化は、クロースインデキシング、ユニフィケーション及びバックトラックの最適化で実現できる。Warrenの命令セットでは、引数は、アーギュメントレジスタと呼ばれるレジスタを介して渡される。パターンマッチング可能なクローズは、第１引数が変数、定数、リスト、構造体のどれであるかにより選択される。その後レジスタ上の引数とクローズのヘッドの引数とのユニフィケーションが行なわれる。このようなレジスタベースの命令はコンパイラの最適化に極めて適している。文献８）で、我々は最適引数によるインデキシング方式と、レジスタ競合を解消し、決定的な組込術語を渡ってレジスタ割り当てを行なうクローズのコンパイル方式について提案した。

以上の最適化を行なってもPrologの非決定性を解消することはできない。これに対し文献10）では非決定的なプログラムを決定的なプログラムに変換する方法により高速化を図っている。

本論文では、バックトラック自体の最適化方式について述べる。バックトラックはシャロウバックトラックとデープバックトラックとに分類される。シャロウバックトラックとは、失敗したクローズに属するプロセジャ内でバックトラックすることを言う。バックトラック情報をセーブ／リストアするWarrenのindexing命令は、デープバックトラック用命令であり、シャロウバックトラックに対しては効率が悪い。このため、まずWarrenの命令セットの枠内で効率の良いシャロウバックトラックを実現する方式について述べる。更にモード情報を用いた新しい最適化方式の提案を行ない、最後に内蔵型Prologプロセッサ IPP[8,9]による性能評価結果について述べる。

2. Prolog命令セット

IPPの命令セットは、Warrenの命令セットを拡張したProlog命令セットと、汎用の命令セットからなる。以下の議論ではIPPの命令セットを用いることにする。

Prolog命令は、get, put, unify, procedural, indexing, 及びbuilt-in命令に分かれる。

アーギュメントレジスタとヘッドの引数とのユニフィケーションを行なうget命令、及びボディゴールの引数をアーギュメントレジスタにロードするput命令は、Warrenの命令とほぼ同じである。Warrenのunify命令に加えて、固定長のリストの要素とユニファイする命令を加えた。

indexing命令に属するswitch-on-term命令は、procedural命令であるexecute及びcall命令と組合せ、更に最適引数でインデキシングできるようにアーギュメントレジスタをオペランドとして加えた。即ち

$$\text{execute } A_i, (C_v, C_c, C_l, C_s)$$
$$\text{call } A_i, (C_v, C_c, C_l, C_s)$$

ここでA_iはi番目のアーギュメントレジスタで、C_v, C_c, C_l, C_sは各々Aiが変数、定数、リスト、構造体のときの飛び先アドレスである。

try, try-me-else等のindexing命令は、IPPでは汎用命令とProlog命令

undo X

で実現した。上記の命令は、トレイルスタックに格納されたアドレスXからトレイルのトップまでに格納された変数についてundo処理を行ないかつトレイルを開放する。

built-in命令は次の組込術語に対応する命令である。

(1) cut及びif-then-else
(2) 算術演算
(3) 比較演算
(4) var, integerなどの型チェック

3. シャロウバックトラックの高速化

プロセジャ呼び出しを決定的にするために、入力及び入出力となる引数の中からインデキシングを行なう最適引数を選択する方法が考えられる。しかしこのような最適化をしても、バックトラックの可能性は残る。

シャロウバックトラックの高速化のために考えるべきことは以下の3点に集約される。

(1) シャロウバックトラックができるために必要十分なバックトラック情報を決定する。
(2) ユニフィケーションの失敗ができるだけ早く見つかるようにユニフィケーションの順序を決定する。
(3) 共通のユニフィケーションを検出し、シャロウバックトラック時にそれを除く。

この節では、シャロウバックトラックを実現するための枠組について議論する。この節より後でスタック管理及びモード情報を用いた最適化方式について述べる。

バックトラックを実現するためのindexing命令には次のものがある。

try C	try_me_else C
retry C	retry_me_else C
trust C	trust_me_else fail

try命令は、バックトラックに必要な情報を持ったチョイスポイントをローカルスタックに格納する。チョイスポイントの内容は以下のとおりである。

(1) 別解へのポインタ (ACP)
(2) アーギュメントレジスタ (A_1〜A_m, m：引数個数)
(3) top-of-heapレジスタ (H)
(4) top-of-trailレジスタ (TP)
(5) ラストチョイスポイントレジスタ (B)
(6) continuation program pointer レジスタ (CP)
(7) ラストエンバイロンメントレジスタ (E)

チョイスポイント格納後、オペランドに指定されたクローズとのユニフィケーションを行なう。もしユニフィケーションに失敗すると、try命令の直後の命令を実行する。retry命令は、ラストチョイスポイントから上記のレジスタの回復処理を行ない、以降try命令と同様の処理を行なう。trust命令は、別解のクローズの最後のクローズについて用いられる。ラストチョイスポイントをスタックからポップアップする以外はretry命令と同じである。try-me-else, retry-me-else命令ではオペランドは、別解のアドレスを指定する。

チョイスポイントに格納される情報は、デープバックトラックのためには必要十分な情報であるが、シャロウバックトラックするには余分な情報がある。get命令の実行の後にアーギュメントレジスタの内容は、undo処理をすることにより回復できる。従ってget命令でユニファイされたアーギュメントレジスタはセーブする必要がない。(put, unify命令ではこのことは成立しない。)従ってget命令で破壊されるアーギュメントレジスタの内容は回復しなくてよい。またCPは、シャロウバックトラックによって破壊されないから回復は不要である。バックトラックは、チョイスポイントに格納されたACPから開始されるので、シャロウバックトラックは、ACPを書き替えることによって制御できる。ここで次のプログラムを考える。

```
p(a,b).
p(a,c).
p(a,d).
```

第1引数でインデキシングされているとすると、高速なシャロウバックトラックを実現するコードは次のようになる。

```
    try d1
e1: retry d2
e2: trust d3

d1: movea s2, ACP
    get_constant a, A1
    get_constant b, A2
    movea e1, ACP
    proceed

s2: undo TR_B
    movea s3, ACP
d2: get_constant a, A1
    get_constant c, A2
    movea e2, ACP
    proceed

s3: undo TR_B
    move B_B,B
d3: get_constant a, A1
    get_constant d, A2
    proceed
```

上記コードにおいてmovea s2, ACPは、s2のアドレスを、ACPにロードすることを意味し、TR_Bはラストチョイスポイントに格納されたTRの値である。

try命令で作られるチョイスポイントは、デープバックトラックのためである。この命令ではACPは退避する必要はない。ラベルd1でACPがシャロウバックトラックのスタートアドレスs2に書き替えられるためである。ユニフィケーションに成功するとACPは再びデープバックトラックに備えてe1に書き替える。ユニフィケーションに失敗すると、シャロウバックトラックが始まり、実行は、ラベルs2から再開される。変数は、undo処理され、ACPはs3に書きかえられる。シャロウバックトラックの最初では、try、あるいはretry命令を実行しなければならないが、その後のシャロウバックトラックでは、これらの命令の実行は不要となり、高速化が可能となる。

デープバックトラックが決して起こらない場合も生じる。k個のクローズからなるプロセジャが次のように与えられるとする。

```
h(H^1_1,..., H^1_n):-    b_in^1,    !, b^1(B^1_1,...B^1_m1),...
h(H^2_1,..., H^2_n):-    b_in^2,    !, b^2(B^2_1,..., B^2_m2),...
                  .
                  .
                  .
h(H^k-1_1,..., H^k-1_n):- b_in^k-1, !,
                     b^k-1(B^k-1_1,...,B^k-1_m(k-1)),...
h(H^k_1,..., H^k_n):-     b_in^k,    b^k(B^k_1,...,B^k_mk),...
```

ここでb-in^i(i＝1, …, k)は組込命令である。簡単のためそれらの引数は省略してある。(以下の議論では、上記をプロセジャの一般形として用いるが、カットは最初のk-1個のクローズに含まれる必要はない。)カットがb-in^i(i＝1, …, k-1)にあるため、デープバックトラックは起こらない。このように決定的な組込述語を命令として定義することにより、シャロウバックトラックの定義は、ヘッドとそれに続く組込述語命令まで含めて拡張することができる。この種のプロセジャをシャロウバックトラックプロセジャと呼ぶことにする。このプロセジャではチョイスポイントの生成は不要となる。先の例がシャロウバックトラックプロセジャとすると、コードは次のようになる。

```
    move ACP, A3
    movea s2, ACP
    move TR, A4
    get_constant a, A1
    get_constant b, A2
    move A3, ACP
    proceed

s2: undo A4
    movea s3, ACP
    get_constant a, A1
    get_constant c, A2
    move A3, ACP
    proceed

s3: undo A4
    move A3, ACP
    get_constant a, A1
    get_constant d, A2
    proceed
```

最初ACPをA3にセーブし、ACPをシャロウバックトラックのアドレスs2に書き替える。ユニフィケーションに成功すると、A3の内容をACPに回復する。このコードは先のコードより更に効率が良くなる。

Prologの一つの特徴は、引数の入出力が決まっていないことである。しかしながら一般的には、プロセジャコールの如何に係わらず多くの引数は入力あるいは出力に固定される。このモード情報は、シャロウバックトラックの最適化に有用な情報である。モード情報は、モード宣言あるいはモードの推論[11]によって得られるが、ユニフィケーションの失敗を起こすヘッドの引数を限定することができる。従ってこのユニフィケーションに対応するコードをヘッドのコードの最初に持ってくることにより、不要なユニフィケーションすることなく失敗の検出が可能となる。またもしユニフィケーションで変数が拘束されないときは、undo処理も不要となる。以下ではモード情報を用いたシャロウバックトラックの高速化方式について述べる。

4. スタックの管理

ローカル、ヒープ、トレイルの3本のスタックをProlog

の実行に用いる。シャロウバックトラックの途中では、オーバヘッドを削減するためにこれらのスタックの管理を最小限にする必要がある。

4.1 ローカルスタックの管理

ローカルスタックは、チョイスポイントと、エンバイロンメントよりなる。エンバイロンメントは、ボディゴールを渡って現われるパーマネント変数及びゴールの実行のための制御情報を記憶する。シャロウバックトラックプロセジャでは、チョイスポイントは生成する必要がない。エンバイロンメントを生成するか否かはクローズに依存する。またユニフィケーションに失敗すると、その生成がオーバヘッドとなる。従ってエンバイロンメントの生成が必要な時も、ユニフィケーションに失敗する可能性のあるコードを実行してから生成するようにする。

4.2 ヒープの管理

アーギュメントレジスタがヘッドの構造体とユニフィケーションしたとき、レジスタの内容が変数のときは、構造体のコピーがヒープに作られる。出力となる引数を除いて次のどれかの条件が成立したときコピーが作られる可能性がある。

（1）入力となるヘッドの引数が多重のリストあるいは構造体である。

（2）入出力どちらにもなるヘッドの引数がリストか構造体である。

（3）b-in^{i}が“＝”を含みそのどちらの引数も出力モードでなく、入力及び入出力モードの変数が含まれる。

上記の条件が満されないときは、シャロウバックトラック時、top-of-heapレジスタのリセットは不要となる。

4.3 トレイルスタックの管理

入力及び入出力モードに対応するアーギュメントレジスタに含まれる変数は、次のどれかの条件を満たすときに、拘束される。

（1）上記したコピー生成条件をヘッドの引数が満たす。

（2）ヘッドの入力あるいは入出力モードの変数中に同一の変数がある。

（4）b-in^{i}が“is”を含み、左辺の引数が入力引数中のリストあるいは構造体中に現われるか、入出力引数中に含まれる。

もし上記の条件が満たされないときは、シャロウバックトラック中の変数のundo処理は不要である。

5. コードの最適化

5.1 最適化の考え方

シャロウバックトラックを高速化するために、失敗する可能性のあるコードを、代入となるユニフィケーション及びゴール生成の前に生成する。前者のコードをチェックコードと呼び、後者のコードをロードコードと呼ぶことにする。チェックコード及びロードコードの各々で生じるレジスタ競合は、実行順序を変えて解消することとする。両者の間のレジスタ競合は、ワークレジスタを割りあてて解消する。これは不要な代入をシャロウバックトラックの前にしないためである。

チェックコードのアーギュメントレジスタは、リスト、構造体要素のユニフィケーション中に破壊されても、次のクローズで同一のユニフィケーションがあるときは、回復する必要はない。このときはユニフィケーションは最初に行ない後は省略できる。そうでないときはワークレジスタを割りあてて競合解消を図かる。

5.2 有向グラフ化

チェックコードはユニフィケーションで失敗する可能性のあるコードに対応する。クローズiに対して、HU^{i}を次のいずれかの条件を満たす入力あるいは入出力モードの引数番号の集合とする。

（1）j∈HU^{i}に対して、H^i_jは定数のデータ、リストあるいは構造体である。

（2）j, p∈HU^{i}に対して、H^i_jとH^i_pで同一の変数が現われる。

（3）j∈HU^{i}に対して、H^i_jは変数を含み、それがb-in^{i}に現われる。もしb-in^{i}が“is”か“＝”を含むなら、isの左辺あるいは“＝”の両辺が出力変数ではない。

次にH^i_j、j∈HU^{i}に含まれる全ての変数を次のように分類する。

（1）もし変数X及びYが同一のリストあるいは構造体中に現われるならば、XとYは同一の変数の集合G^i_jに含まれる。

（2）もし変数Xが、X以外の変数を含まないリストあるいは構造体に含まれるとき、あるいはXがリスト、構造体中に現われないときは、XはXしか含まない集合G^i_jに含まれる。

次にG^i_j(j＝1，…，$_{P(i)}$)を用いて有向部分グラフDSGU^i_j＝IU^i_j→OU^i_jをチェックコードに対して次のように定義する。

（1）もし変数X∈G^i_jがH^i_p(p∈HU^i)に含まれるならば、p∈IU^i_jとする。

（2）もし変数Y∈G^i_jがY＝B^i_q、q∈B^{i}を満たすならば、q∈OU^i_jとする。但しB^{i}はボディ引数番号の集合で、B^i＝{1，…，m$_i$}。

ここで、BU^i＝OU^i∪… ∪OU$^i_{P(i)}$とする。

5.3 レジスタ競合の解消

まずチェックコードとロードコードとのレジスタ競合解消を考える。もし

$$HU^i \supset BU^i$$

が成立するならば、それらの間での競合はない。もし、j∈BU^i－HU^{i}の要素で

$$j \in H - HU^i$$

を満たすとすると、レジスタ競合が生じる。ここでHはヘッドの引数番号の集合で、H＝{1，…，n}である。j番目のアーギュメントレジスタのかわりにワークレジスタを割りあててレジスタ競合を解消する。従ってそのようなjは、BU^iから削除する。

プロセジャのk個のクローズのレジスタ競合を解消する。これは1～nのアーギュメントレジスタのどれかがi番目のクローズのunify命令で破壊されるときに生じる。i番目のクローズの失敗によるi＋i番目の命令の実行は、i＋1番目のクローズの先頭から実行する必要がある。それはどのステップでユニフィケーションが失敗するか分からないからである。しかしながら2つのクローズでヘッドのユニフィケーションが同じときは、i＋1番目のユニフィケーションをシャロウバックトラック時省略できる。この条件は次のようにして調べる。

全ての$DSGU^q{}_j = IU^q{}_j \rightarrow OU^q{}_j$、q＝i、i＋1に対して、$IU^i{}_j$及び$OU^i{}_j$に対応する引き数が2つのクローズで同一であるならば、それらのクローズ間でのレジスタ競合はない。そうでなければ$p \in BU^i$、$p \in H$に対して、pのかわりにワークレジスタを割りあて、pをBU^iから削除する。

5.4 有向部分グラフの順序づけ

もし有向部分グラフ$DSGU^i{}_p$、$DSGU^i{}_q$の間で、

$$OU^i_q \cap IU^i_p \neq \varnothing \qquad OU^i_p \cap IU^i_q = \varnothing$$

の条件を満たすとき、$DSGU^i{}_p$を$DSGU^i{}_q$より先に実行すれば$DSGU^i{}_p$と$DSGU^i{}_q$の間のレジスタ競合は発生しない。

ロードコードのレジスタ競合解消をするために、チェックコードと同じようにして$H^i{}_p$、$(p \in H - HU^i)$と、$B^i{}_q$ $(q \in B^i - BU^i)$で現われる全ての変数を分類して、$DSG^i{}_j = I^i{}_j \rightarrow O^i{}_j$を作成する。もし有向部分グラフ$DSG^i{}_p$と$DSG^i{}_q$の間で

$$O^i_q \cap I^i_p \neq \varnothing, \qquad O^i_p \cap I^i_q = \varnothing$$

の条件を満たすとき、$DSG^i{}_p$を$DSG^i{}_q$より先に実行すれば、レジスタ競合は発生しない。

5.5 コード生成

ユニフィケーションの失敗をできるだけ早く見つけるコードは次の順序で生成すればよい。

（1）$H^i{}_j$$(j \in HU^i)$中の定数データのユニフィケーション
（2）$DSGU^i{}_j$を決められた順序で展開
（3）命令化された組込述語b-in^iの展開
（4）$H^i{}_j$$(j \in H - HU^i)$中の定数データのユニフィケーション
（5）$DSG^i{}_j$を決められた順序で展開
（6）B^iにおける定数データのロード

5.6 コード生成例

（例1）次のシャロウバックトラックプロセジャを考える。

```
:- mode p(+,-).
p([_], 1):-!.
p([_|_], 2).
```

第1引数は入力でリストであるが、それらの中の引数はボイド変数であるため、第1引数のユニフィケーションでコピーと変数拘束は生じない。

ここで

$$HU^i = \{1\} \qquad i = 1,2$$

有向部分グラフは次のようになる。

$$DSGU^i{}_1 = \{1\} \rightarrow \varnothing \qquad i = 1, 2$$
$$DSG^i{}_1 = \{2\} \rightarrow \varnothing \qquad i = 1, 2$$

従ってこの場合は、チェックコードとロードコード及び2つのクローズ間でレジスタ競合はない。従ってコードは次のようになる。

```
        move ACP, A3
        movea s2, ACP
        get_list A1
        unify_variable  A4
        unify_nil
        get_constant 1, A2
        move A3, ACP
        proceed
s2:     move A3, ACP
        get_list A1
        unify_variable  A4
        unify_variable  A4
        get_constant  2, A2
        proceed
```

（例2）

quick-sortプログラムの次のプロセジャを考える。

```
:- mode split(+,+,-,-).
split(Y, [X|L], [X|L1], L2):- X =< Y, !,
                              split(Y, L, L1, L2).
split(Y, [X|L], L1, [X|L2]):- split(Y,L,L1,L2).
split(_,    [],   [],       []).
```

第2引数でインデキシングすることにより最初の2つのクローズと、3番目のクローズに分離できる。従って最初の2つのクローズのコード生成を考える。“＝<”はbuilt-in命令であるため、シャロウバックトラックプロセジャとなる。

1番目と2番目の引数は、コピーの生成及び変数の拘束がない。従ってtop-of-heapレジスタのリセット及び変数のundo処理は不要となる。

ここで

$$HU^1 = \{1,2\}, \qquad HU^2 = \{2\}$$

であるから$DSGU^i_j$は次のようになる。

$$DSGU^1_1 = \{1\} \rightarrow \{1\},\ DSGU^1_2 = \{2\} \rightarrow \{2\}$$
$$DSGU^2_1 = \{2\} \rightarrow \{2\}.$$

従って、$BU^1 = \{1, 2\}$、$BU^2 = \{2\}$となる。
$HU^1 = BU^1$、$HU^2 = BU^2$であるから、チェックコードとロードコードのレジスタ競合はない。また第1及び第2引数は2つのクローズで同じであるから、クローズ間のレジスタ競合も発生しない。従ってコードは次のようになる。

```
s1:  get_list  A2
     unify_variable  A5
     unify_variable  A2
     move ACP, A6
     movea s2, ACP
     leseq    A5, A1
     get_list   A3
     unify_value  A5
     unify_variable  A3
     move A6, ACP
     execute A2, (fail, s3, s1, fail)

s2:  move A6, ACP
     get_list  A4
     unify_value  A5
     unify_variable  A4
     execute A2, (fail, s3, s1, fail)

s3:  get_nil  A2
     get_nil  A3
     get_nil  A4
     proceed
```

6. 性能評価

IPPと最適化コンパイラとを用いてシャロウバックトラックの高速化方式の評価を行なった。文献13) の4つのベンチマークプログラムを用いて次の3つの最適化方式について評価した。

(1) 文献1) より推定されるWarrenの最適化方式
(2) 最適引数によるイニデキシングと、決定的組込述語を越えたレジスタ割りあて方式[8]
(3) 項(2)+シャロウバックトラックの最適化方式

表1に結果を示す。appendプログラムは決定的であるためシャロウバックトラックの高速化の効果はない。2～4のプログラムはシャロウバックトラックプロセジャを含む。quick-sortプログラムでは、splitがシャロウバックトラックプロセジャとなり、今まで述べた全ての最適化方式が適用可能となり大幅な性能向上が実現できた。しかしながら8-queenではその効果はあまりでていない。それはシャロウバックトラッププロプセジャgenerateがあまり呼ばれないからである。

表1　最適化の効果と性能

No.	プログラム	方式1	方式2	方式3	kLIPS
1	append	1	1	1	1125
2	q-sort	0.84	1	2.0	921
3	8-queen (first)	0.36	1	1	1137
4	8-queen (all)	0.37	1	1	1081

方式1：Warrenの方式
方式2：最適引数によるインデキシング＋大域レジスタ割当て
方式3：方式2＋シャロウバックトラックの最適化

7. 結果の検討

シャロウバックトラックの最適化のためにモード情報を用いたが、それなしでも最適化は可能であり、また効果も大きい。それはtry、retry、trust命令が多くのメモリアクセスを必要とするからである。Prologで書かれたIPPのPrologコンパイラを解析した結果、98個のプロセジャのうち、27個のプロセジャがインデキシングで決定的となり、48個のプロセジャがシャロウバックトラックプロセジャとなった。従って実際のプログラムでも、シャロウバックトラックの最適化による効果は大きいものと考えられる。

8. 結　論

WarrenのProlog命令セットをベースにして、モード情報を用いたシャロウバックトラックの高速化方式について述べた。新しく開発された手法は以下のとおりである。

(1) ヒープのリセット、変数のundo処理、アーギュメントレジスタの回復処理等、不要なバックトラック情報の削除。
(2) 失敗ができるだけ早く見つかるようにユニフィケーションの順序の変更
(3) プロセジャにおける共通のユニフィケーションの検出と、シャロウバックトラック時での省略

これらの手法は、バックトラック情報のセーブ／リストアが不要なシャロウバックトラックプロセジャに対して効果が高い。本手法の有効性は、内蔵型Prologプロセッサ IPPを用いて確かめられ、quick-sortプログラムで2.0倍、0.9MLIPSの性能が実現できることが確かめられた。

参考文献

1) D.H. Warren, "An Abstract Prolog Instruction Set," Technical Note 309, Artificial Intelligence Center, SRI International, October 1983.
2) R. Nakazaki et al., "Design of a High-speed Prolog Machine(HPM)," Proceedings of the 12th International Symposium on Computer Architecture, June 1985, pp 191-197.
3) T. P. Dobry et al., "Performance Studies of a Prolog Machine Architecture," ibid., pp 180-190.
4) A.M. Despain, "A High Performance Prolog Co-processor," Proceedings of WESCON 85, 1985, No 18/2.
5) M. Yokota et al., "The Design and Implementation of a Personal Sequential Inference Machine: PSI," New Generation Computing Vol. 1, pp 125-144, 1983.
6) T. Kurokawa et al., "A Very Fast Prolog Compiler on Multi Architecture," Proceedings of Fall Joint Computer Conference, October 1986, pp 963-968.
7) S. Arai, "The Design of Prolog Compiler," Proceedings of the First Annual Conference of JSAI '87 , June 1987, pp185-188.
8) S. Abe et al., "High Performance Integrated Prolog Processor IPP," Proceedings of the 14th International Symposium on Computer Architecture, June 1987, pp 100-107.
9) S. Yamaguchi et al., "Architecture of High Performance Integrated Prolog Processor," Proceedings of Fall Joint Computer Conference, October 1987, pp 175-182.
10) 松本他，“PROLOGの非決定性を最適化するコンパイル方式の評価”，情報処理全国大会 6H-7，昭和63年3月，pp803-804
11) S. K. Debray and D. S. Warren, "Automatic Mode Inference for Prolog Programs," Proceedings of 1986 Symposium on Logic Programming, September 1986, pp 78-88.
12) S. Abe et al., "A New Optimization Technique for a Prolog Compiler," Proceedings of Compcon 86 Spring, March 1986, pp 241 -245
13) 奥野，“第３回LISPコンテスト及び第１回PROLOGコンテストの議題案”，情報処理学会記号処理研究会28-4，昭和59年6月

盛光印刷所