

PSI-IIの機械命令セット評価

立野裕和 * 近藤誠一 * 中島浩 * 中島克人 **

*: 三菱電機株式会社

**: 新世代コンピュータ技術開発機構 (ICOT)

第五世代コンピュータ・プロジェクトの一貫として、マルチPSIの要素プロセッサおよびフロント・エンド・プロセッサとして用いられるPSI-IIを開発した。

PSI-IIはWAMをベースとし、各種最適化命令を導入した機械命令セットを使用している。機械命令セットには複数の機械命令を統合した複合化命令や本稿で示すクローズ・インデキシング技法である "if-then-else", "neck-cut"最適化のための命令が含まれている。これらの機械命令とクローズ・インデキシングの効果をベンチ・マークプログラムおよび大規模ソフトウェアに対し測定することで、我々の設定した機械命令セットの妥当性を示す。

Evaluation of instruction set of the PSI-II

Hirokazu Tateno * Seiichi Kondoh * Hiroshi Nakashima * Katuto Nakajima **

*: Mitubishi Electric Corporation

5-1-1, Ofuna, Kamakura 247, Japan

**: Institute for New Generation
Computer Technology

1-4-28, Minato-ku, Tokyo 108, Japan

PSI-II is the element processor of the parallel inference machine called Multi-PSI. PSI-II is also used as a stand-alone sequential inference machine.

The machine instruction set of the PSI-II was designed based on WAM, and many optimization was employed in it such as "merged instruction", "if-then-else" and "neck-cut". Merged instructions are to perform functions of two or more basic instructions, and are expected to execute faster than by these separate instructions. If-then-else and neck-cut are one of the clause indexing techniques for the predicates where conventional indexing techniques cannot be applied.

This paper described some evaluation results for the optimizations in our machine instruction set with some benchmarks and large scaled practical programs

1. はじめに

我々は第五世代コンピュータ・プロジェクトの一貫として、マルチPSIの要素プロセッサおよびフロント・エンド・プロセッサとして用いられるPSI-IIを開発した。

PSI-IIの機械命令セットはWAM[1]をベースとしさらに多数の最適化命令を導入することでKLOプログラムの高速実行を目指している。

本稿では、PSI-IIの機械命令セット設計時に前提となるPSI-IIのアーキテクチャについて簡単に述べた後、機械命令セットの内特徴的な"if-then-else"最適化、"neck-out"最適化の効果および複合化命令の効果について述べる。最後に大規模プログラムのコンパイル・コードに対する静的/動的解析結果について報告する。

2. PSI-II アーキテクチャの特徴

PSI-II では、KLO をコンパイラによって機械語にコンパイルする方式を採用している。機械語は WAM で提唱されたレベルのものを採用しており、各機械語はマイクロ・プログラムによって直接実行される。ハードウェアの設計にあたっては[2,3,4]、機械命令の設計の自由度を最大にするために水平型マイクロ・プログラム制御方式を採用した。マイクロ・プログラムは1語 53 ビット長からなる。基本的には、ALU 演算、フラグ/カウンタ操作、2方向条件分岐/無条件分岐/多方向分岐、メモリ・アクセスの4操作を同時に実行することが出来る。またハードウェアによるパイプライン制御は行っていないが、マイクロ・プログラム制御による3段の命令パイプライン機構をもっている。これによって最小1ステップで1機械命令を実行することが出来る。

KLO には算術、論理演算系、比較系、構造体操作系、データ・タイプチェック等の一般的な組込述語の他に KLO の特徴である実行順序制御系、ESP サポート系、OS 機能をサポートするためのシステム制御系等の多くの組込述語がある。これらはすべて機械命令として準備され、マイクロ・プログラムによって直接実行される。

データ表現形式は 8 ビットタグ (6 ビットでデータ・タイプを表わし 2 ビットはガベージ・コレクション用である) と 32 ビットの値部からなる 40 ビット1語構成である。

機械語命令はバイトコードとはせずに1語単位としている。機械命令、オペランドの形式を図1に示す。

39 3433 2423 0

001111	op-code	operand
--------	---------	---------

オペランドの形式

23 16 15 8 7 0

operand 1	operand 2	operand 3
-----------	-----------	-----------

operand 1	operand 2
-----------	-----------

39 3231 2423 1615 87 0

tag	op1	op2	op3	op4
-----	-----	-----	-----	-----

39 3231 0

tag	operand
-----	---------

図1 機械命令・オペランドの形式

機械語の値部は 8 ビットづつに 4 分割され、タグの下位 2 ビットと値部上位 8 ビットの合計 10 ビットを命令コードとしている。

8 ビットの各オペランドは 32 個準備されている引数レジスタのレジスタ番号を表わす場合、または即値や述語呼び出しなどの分岐命令のための相対アドレスとして使われる。

16 ビットのオペランドは即値、または分岐命令のための相対アドレスとして使われる。

複数語から成る機械命令の場合、命令語の2語目を 8 ビットづつ 4 分割し上位 16 ビットでそれぞれ引数レジスタのレジスタ番号を表わす。またはタグ付きの即値として使用される他 16 ビットずつ 2 分割し分岐命令のための相対アドレスとして使われる。

3. "if-then-else" 最適化、

"neck-out" 最適化の効果

3.1 "if-then-else" 最適化

"if-then-else" 最適化は PSI で導入された最適化技法であり [5]、図2に示すような従来のクローズ・インデクシングが不可能であるような述語の高速実行のために用いられる。

PSI の場合 "if-then-else" 最適化の導入の主目的は SIMFOS (PSI, PSI-II のプログラミング・オペレーティング・システム) の高速化であった。そのため、SIMFOS 開発者に対し特別なシンタックスを提示し、プログラム・ソースを書きなをすことによって導入した。

PSI-II においてはコンパイラの最適化技法として採用しており、PSI で導入した特別なシンタックスを用いていなくても "if-then-else" 最適化の条件を満たす述語にたいしては "if-then-else" のコードが生成される。

```
p(X,Y):-integer(X),X>Y,! ,q(X,Y),s(Y)
```

↓
+ 条件部

```
p(X,Y):-r(X,Y),s(Y)
```

図2 "if-then-else" 最適化の対象となる述語例

図2に示した述語例において下線を施した部分を条件部とみなすことが出来る。この条件部の内 `integer(X)` および `X>Y` はそれぞれ組込述語である。これらの組込述語の内 `integer` は引数が整数であれば成功する。また、`X>Y` は第1引数が第2引数より大きい場合に成功するものである。条件部の実行が成功するとカットにより述語 `p` の `backtrack` フレームは不要となる。さらに、条件部の実行が終了するまでの間に述語 `p` の呼び出し側変数に対する値の束縛、述語 `p` が呼びだされた時点での引数レジスタの破壊が行われていない。

したがって条件部の実行結果によってつぎに実行する命令を決定するような2方向分岐命令を導入することによって、述語 `p` のための `backtrack` フレームの生成を省略することが可能であり、ヘッド引数に着目したクローズ・インデキシングが不可能な述語においても高速な処理が可能となる。

図2に示したような述語の特徴を整理し "if-then-else" 最適化の適用条件を表1にまとめる。

表1 "if-then-else" 最適化の適用条件

- (1) オルタネート・クローズが1以上存在する。
- (2) ヘッド引数がすべてことなる。
- (3) ボディーにカットがあり、そのカットの実行以前に呼び出し側の未定義変数に値を束縛したり、不正入力などの例外事項が発生しない組込述語から条件部が成っていること。

PSI-II においてクローズ内 OR を図3に示すように展開している。このため、表1に示した "if-then-else" 最適化の適用条件はORの左の枝が条件(3)を満たすだけで良い。

```
p(X,Y):- (X>Y,! ,q(X,Y) ; r(X,Y)),s(Y) .
```

```
p(X,Y):- $or-p(X,Y),s(Y).
$or-p(X,Y):- X>Y,! ,q(X,Y).
```

↓
+ 条件部

```
$or-p(X,Y):- r(X,Y).
```

図3 クローズ内ORの展開

```
jump-unless-less-than X1 X2 Else
execute q/2
Else:
execute r/2
```

図4 図3のOR部分のコンパイル結果

図4に図3の OR 部分のコンパイル結果を示す。条件部の組込述語は `jump-unless-less-than` 命令にコンパイルされる。この命令では、比較結果が失敗であれば単に指定されたアドレスに分岐する。この様に、比較を行う組込述語や、データ・タイプを調べる組込述語で不正入力などの例外事項を検出しないものについて実行結果に応じた2方向分岐を行う命令を導入し以下に示す処理を省略し、高速なクローズ・インデキシングを実現している。

- (1) `backtrack` フレームを生成しない。
- (2) (1)により `fail` 処理を省略。

3.2 "neck-out" 最適化

"if-then-else" 最適化は条件部の実行後その実行結果が成功であっても失敗であっても、呼びだされた述語の `backtrack` フレームが不要となること。および、条件部の実行終了までに呼び出し側の未定義変数に値を束縛しないことから `fail` 時に行われる `undo` 処理を行わなくても述語が呼びだされた時点の実行環境が破壊されない点を利用した最適化技法であった。

これに対し "neck-out" 最適化は、呼びだされた述語のクローズ選択が終了するまでにユーザー述語の呼び出しがないこと。およびクローズ選択終了時点で呼びだされた述語の `backtrack` フレームが必ず不要になる場合に適用され、呼び出し側の未定義変数に値を束縛することは許される。正確な適用条件を表2に示す。

表1に示した "if-then-else" 最適化の適用条件に比べて緩い条件となっているが適用時の速度向上は "if-then-else" 最適化よりもいくぶん低くなる。

表2 "neck-out"最適化の適用条件

- (1) 最後のクローズを除くすべてのボディー部にカットがあり、カットの前にユーザー定義述語の呼び出しがないこと。
- (2) 各クローズのカット命令以前のユニフィケーション処理において引数レジスタを破壊することなくコンパイルが可能なこと。

図5に "neck-out" 最適化の適用可能な述語例を示す。図5に示したような述語に対しては `switch-on-term` 命令等によるクローズ・インデキシングが行えないために通常は `try`, `retry`, `trust` 等の命令で逐次実行される。

しかし最後のクローズを除きカットが実行される以前にユーザー定義述語の呼び出しがなく、クローズ選択が終了した時点で呼びだされた述語の backtrack フレームは必ず不要となる。この点を利用して以下の処理方式をとることによって高速化を図るのが "neck-cut" 最適化である。

- (1) backtrack フレームをスタック上に生成しない。
- (2) カットの実行までに必要な backtrack 情報を専用のレジスタ上で保持する (PSI-II では、変数のトレール情報は通常のトレール・スタック上でやっている)。

backtrack 情報を専用のレジスタ上で保持している状態を fast-mode と称している。このモードへの切り替え、次クローズ選択、fast-mode 終了のために fast-try, fast-retry, fast-trust 命令等を導入した。また cut-me 命令においては fast-mode であるかどうかによって backtrack 情報のキャンセルの方式を切り替えるように仕様変更を行った。

```
p(X,X,Z):-!,q(X,Z).
p(X,Y,Z):-integer(Y),10= X + Y,!,s(Z).
p(X,Y,Z):- t(X,Y,Z).
```

図5 "neck-cut" 最適化の対象となる述語例

fast-try	Next/3
get-variable-t	X04 A01
get-value-t	X04 A02
cut-me	
put-value-t	X03 A03
execute	q/2
Next:	
fast-retry	Last
integer	A02
add	A01,X02 X04
get-integer	10 X03
cut-me	
put-value-t	A03 A01
execute	s/2
Last:	
fast-trust	
execute	t/3

図6 図5のコンパイル結果

図5に示したように fast-mode 中に実行される組込述語の中には不正入力等の例外事項を検出するものも許される (10=X+Y は組込述語 add にコンパイルされるが add は不正入力などを検出する)。例外事項が検出された場合には [6] ファーム・ウェアによってあらかじめシステムに登録されている例外処理用の述語を呼びだす。この処理が行われると、カットを実行するまでにユーザー述語の呼び出しが起きてしまい、"neck-cut" 最適化の適用条件を破ることとなる。そのため、例外事項が検出された時点で fast-mode であれば例外処理用の述語を呼びだす前に通常の backtrack フレームを生成し fast-mode をリセットするようにしている。

3.3 最適化による速度向上

ここまで述べてきた最適化によるベンチマーク・プログラム (nrev30, qsort50, 8queen) に対する最適化の効果を表3に示す [7]。ベンチマーク・プログラムをコンパイルする際に適用する最適化レベルを表4にまとめて示す。

表3 ベンチマーク・プログラムの実行性能

最適化レベル	nrev30	qsort50	8queen
レベル0	1.53	3.86	29.0
レベル1	1.53	3.86	29.0
レベル2	1.53	3.01	21.8
レベル3	1.53	3.01	21.8

注) マシンサイクルは166.7nsec, 測定単位はmsec

表4 コンパイラの最適化レベル

最適化レベル	最適化技法	
	"if-then-else"	"neck-cut"
レベル0	適用せず	適用せず
レベル1	適用	適用せず
レベル2	適用せず	適用
レベル3	適用	適用

表3に示す様に qsort50, 8queen に対して "neck-cut" 最適化の効果が現れている。この最適化によって qsort50, 8queen はそれぞれ 22%, 24% の高速化が達成された。しかしこれらのプログラムは "if-then-else" 最適化の適用条件を満たさないために "if-then-else" 最適化の効果をみることは出来ない。

"if-then-else" 最適化の効果をみるために qsort50 の partition の第2, 第3クローズをクローズ内 OR に書き換え "if-then-else" 最適化の対象となるようにした。以下に書き換えた partition のプログラムおよび prolog コンテストのプログラムを示す。

original: prologコンテストのプログラム。

```
partition([X2|List],X1,[X2|List1],L):-  
  X2<X1,! ,partition(List,X1,List1,L).  
partition([X2|List],X1,L1,[X2|List2]):-  
  partition(List,X1,L1,List2).
```

qsort-A: クローズ内CRに変更。

```
partition([X2|List],X1,L1,L2):-  
  (X2<X1,! ,L1=[X2|List1],  
    partition(List,X1,List1,L2);  
  L2=[X2|List2],  
    partition(List,X1,L1,List2)).
```

qsort-B: PSIで導入した"if-then-else"用の
シンタックスに変更

```
partition([X2|List],X1,L1,L2):-  
  (X2<X1 L1=[X2|List1],  
    partition(List,X1,List1,L2);  
  L2=[X2|List2],  
    partition(List,X1,L1,List2)).
```

表5に各プログラムの実行性能向上率を示す。なを
実行性能向上率の元になるデータとしては prolog
コンテストのプログラムをレベル0でコンパイルし
た場合の実行性能をとった[8]。

表5 "if-then-else"最適化による
qsortの実行性能向上率

プログラム	実行性能向上率	最適化レベル
original	28%	レベル2
qsort-A	44%	レベル1
qsort-B	51%	レベル1

"neck-cut"最適化に比べ "if-then-else"最適化
の効果が高い。これは "if-then-else"最適化がい
わば従来から行われてきた単純な2方向分岐命令を
用いているのに対し "neck-cut"最適化は簡素化し
たとは言えpartitionの各クローズを逐次に行
っているためである。

qsort-A, qsort-Bの各プログラムの性能向上率の
違いについて述べる。qsort-Aの場合、クローズ内
CRはいったん別クローズに展開されるため以下の述
語と等価である。

```
partition([X2|List],X1,L1,L2):-  
  $or-partition(X2,List,X1,L1,L2).
```

```
$or-partition(X2,List,X1,L1,L2):-  
  X2<X1,! ,L1=[X2|List1],  
    partition(List,X1,List1,L2).
```

```
$or-partition(X2,List,X1,L1,L2):-  
  L2=[X2|List2],  
    partition(List,X1,L1,List2).
```

この様に展開された述語を呼びだすために put/
get等のオーバーヘッドが必要になる。

これに対し qsort-Bの場合、クローズ内CRの部
分を別クローズに展開せずに、呼びだされた
partitionと同じ呼び出しレベルで実行するために、
先に述べた qsort-Aの場合にあったようなオーバー
ヘッドは存在しない。それによって、qsort-Aと比
較した場合、より高速に実行されている。

しかしこの様な形に "if-then-else"最適化の対
象となりうるクローズ内CRをコンパイルしてしま
うと、クローズ内CRの呼び出しレベルに混乱が生
じ、カット等の有効範囲に関して問題が生じるた
めにここで示した特殊なシンタックスで記述され
ている場合にのみこの様なコンパイル・コードを生成
するようにしている。

4. 複合化命令の効果

4.1 複合化命令

一般にプログラムのコンパイル結果を調べると、
特定の命令の連続したパターンが頻繁に現れるこ
とがある。この特定の命令の連続したパターンを一つ
の命令にしてしまったものが複合化命令である。
PSI-IIの場合各機械命令は水平型のマイクロ・プロ
グラムで記述しているために複数の命令を複合化す
ることによりマイクロ・プログラムの並列動作機能
が単一命令が連続している場合よりもより有効に機
能することが期待できる。

すでに2で述べたように PSI-IIはワードマシン
である。各機械語は24ビットのオペランドをもっ
ている。例えばオペランドとして8ビットしか使用
していない機械命令の場合、オペランド16ビット
は無駄になっている。この様な機械命令2命令を複
合化することが出来れば今まで2ワード必要であ
った命令語が1ワードで済むこととなりコード量削減
の効果が期待される。

PSI-IIの機械命令セットにおいて準備した複合
化命令は以下のものである[9]。

1) get-list 命令 + 変数 unify 命令。

Car, Cdrともに変数であるようなリストのユニ
フィケーションは get-list 命令に続いてリス
ト要素の変数の種類に応じた2個の unify 命令
によって成される。この場合、SR (ストラクチャ・
レジスタ)を介した処理方式となる。しかし
処理対象となる主記憶は必ずアドレスが連続した
領域であるため、ハードウェアでサポートされて
いる主記憶への連続アクセス機能が有効に使用す
ることが出来る。そこで、get-list命令とこれに
続く2個のunify命令を複合化した命令を準備し
た。準備すべき命令の数はリスト要素の変数の組
み合せで決まる。変数の種類は一時変数/永久変
数およびその変数が初出であるかないかに応じて
4種類になる。そのため、2個の unify 命令列
は合計16種類になる。しかし我々はこれらの内
出現頻度の高いと思われる5種類の unify 命
令列を複合化した命令を準備している。

2) execute 命令 (call 命令)

+ インデキシング命令

execute 命令または、call 命令 (いずれも無条件分岐命令) によって呼びだされた述語がデータ・タイプによるインデキシングが可能な場合、呼びだされた述語の最初に実行される命令は必ず switch-on-term 等のインデキシング系の機械命令 (条件分岐命令) である。このような場合、2回連続して分岐命令が実行されることになる。このようなことが起きると命令フェッチをパイプライン化していることに意味がなくなってしまう。そこで我々は execute 命令のような述語呼び出し系の命令と switch-on-term 命令を複合化した命令を準備し、2回連続して実行される分岐命令を1回ですむようにした。

3) execute 命令 + deallocate 命令

実行環境の生成を必要とする述語のラスト・ゴールを呼び出す場合、TRO をかけるために実行環境を解放する deallocate 命令が必ず実行される。このため execute 命令と deallocate 命令を複合化した命令、さらに execute 命令が switch-on-term 命令と複合化できる場合もあるため execute-with-deallocate-and-switch 命令も準備している。

4) proceed 命令 + deallocate 命令

PSI-II において組込述語は機械命令によって直接実行される。そのため述語のラスト・ゴールが組込述語の場合、そのような述語の最後は proceed 命令によってその述語の呼び出しもとにリターンすることになる。仮にこの様な述語が実行環境を生成していたとすれば先に述べた deallocate 命令も必ず実行される。そこで我々は proceed 命令と deallocate 命令を複合化した命令を準備している。

5) cut 命令 + proceed 命令

(proceed-with-deallocate 命令)

cut 命令も組込述語の1種類であり述語の最後に現れた場合は先に述べたように proceed 命令によって述語のよびだし元にリターンすれば良い。しかし述語の最後に cut 命令が現れる可能性が他の組込述語に比べ高いと思われるために cut 命令に proceed 命令 (proceed-with-deallocate 命令) を複合化した命令を準備している。

4. 2 速度向上

プログラム実行時の命令実行頻度を調べることで複合化命令導入時に期待された実行速度の向上がどの程度なされているかを調べた。測定には PSI-II 上でファイルのエディタへの読み込みおよび読みだしたファイルのコンパイルとディスクへのセーブを行うプログラムを作りその実行時間と命令実行頻度を測定した。

実行性能向上率は複合化命令のマイクロ・プログラム上のステップ数と複合化した各命令のステップ数の差をとり実際に実行された複合化命令の数から複合化したことによるステップ数を求めて計算した値である。

表 6 複合化命令の効果

プログラムの実行時間 (msec)	15100
実行命令総数	12257862
複合化命令実行総数	1149246
実行性能向上率	約 1.2%

4. 3 コード削減量

複合化命令を導入することによってコンパイル・コード量の削減が期待される。これをみるために PSI-II の OS である SIMPOS のコンパイル・コードに対して機械命令ごとの静的出現頻度を求め複合化命令の導入によるコード量削減効果をみた。

一般に複数の命令を1個の命令にまとめることによってコンパイル・コードは削減される。しかし先に紹介した複合化命令の内 execute-and-switch 命令等は各呼び出し側に switch 命令があり同時に呼びだされる側にも switch 命令があるために全体のコード量としては増加することになる。

測定結果は複合化命令を導入したことにより全体で約2%程度のコード量削減が成されていることが判った。

5. 大規模プログラムを用いた静的/動的解析

5. 1 評価用プログラム

大規模プログラムにおける機械命令の出現頻度をみることによって本稿で述べてきた複合化命令やコンパイラの最適化技法の有効性をみるために、以下のプログラムを対象として命令出現頻度および最適化技法の違いによる速度向上比を測定した。

1) SIMPOS

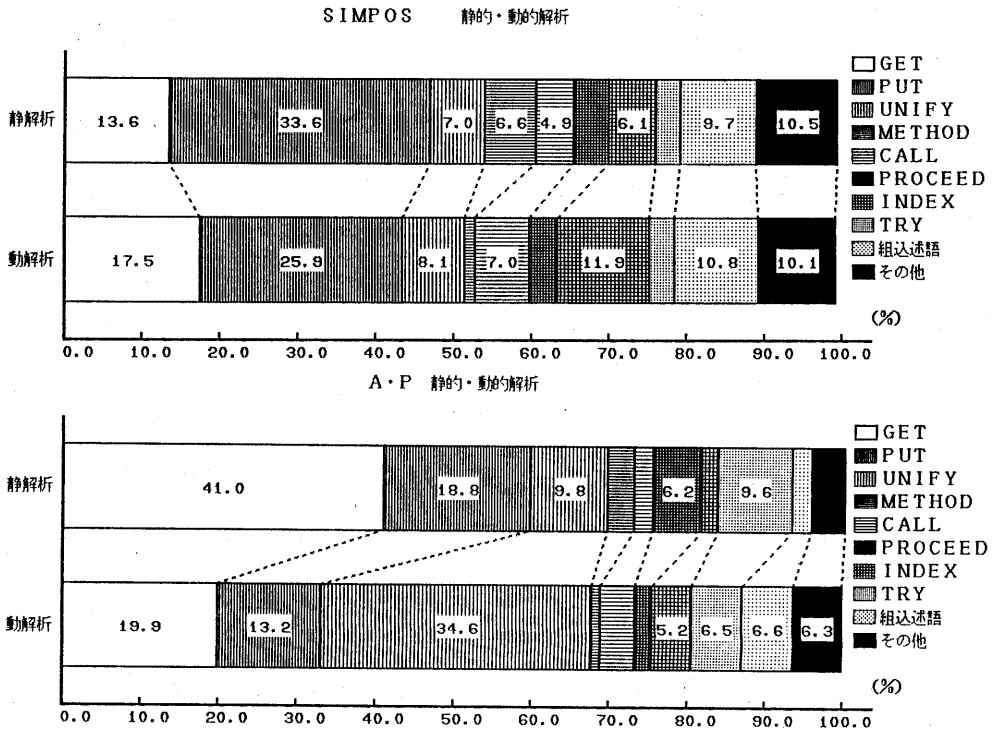
SIMPOSが提供しているプログラム開発環境のうちエディタによるファイルの読込、読込んだファイルのコンパイルとディスクへのファイルセーブをプログラムから動作するように作成し SIMPOS の動的解析を行った。また評価対象とした SIMPOS は本稿で述べた "neck-cut", "if-the-else" の最適化を施したものである。

2) アプリケーション (A・P)

自然言語処理に関連する大規模アプリケーションプログラムを一例として取り上げた。

5.2 命令出現頻度

図7にSIMPOSのコンパイル・コードの静的解析結果および先に述べたプログラムによる動的解析結果を示す。また図8にA・Pのコンパイル・コードの静的解析結果および動的解析結果を示す。



GET: get系機械命令

UNIFY: unify系機械命令

CALL: ESPにおけるローカル述語呼び出し

INDEX: switch-on-term等のいわゆるインデキシング系命令および本稿で述べた"if-then-else", "neck-cut"最適化のための機械命令。

その他: allocate命令, deallocate命令, jump命令, fail命令等。

PUT: put系機械命令

METHOD: ESPのメソッド呼び出し。

PROCEED: proceed命令。

TRY: 通常のTRY, RETRY等の機械命令。

組込述語: 組込述語。

SIMPOSの静的/動的命令出現頻度の特徴を以下に列挙する。

- 1) 静的/動的どちらの場合もget系, put系, unify系の命令の出現頻度が60%程度である。
- 2) 静解析の場合OSのすべてのコードを調べているためにメソッド呼び出しとローカル呼び出しがほぼ同じ割合で現れている。
- 3) 動的解析によれば、命令実行頻度としてはインデキシング系命令の実行頻度はTRY系命令の実行頻度の約5倍程度あり、本稿で述べたような各種インデキシング手法が有効であると言える。

A・Pの静的/動的命令出現頻度の特徴を以下に列挙する。

- 1) 静的/動的どちらの場合もget系, put系, unify系の命令の出現頻度が70%程度である。特に静解析においてget系の命令出現頻度が高いのは、今回測定したA・Pにおいて参照するデータ・ベース的な情報をユニット・クローズとしてプログラム中に保持しているからである。
- 2) 今回測定したA・Pの場合実行時にunify系命令の実行頻度が高い。これはこのプログラムにおいてしばしば使用される情報をリストで保持しているためである。
- 3) インデキシング系命令とTRY系命令の実行比がほぼ等しくなっておりこのA・Pにたいして本稿で述べたインデキシング技法はあまり効果をもたないと思われる。

複合化命令の内 get-list と get-XX-YY-list に関して動的命令出現頻度について注目し整理した。表 7 に get-list 系命令の出現頻度を示す。

表 7 get-list 系命令の出現頻度

機械命令	実行頻度
get-list	48%
get-vart-vart-list	26%
get-vart-varp-list	10%
get-valt-valt-list	1%
get-valt-varp-list	6%

注)

リスト要素の変数が一時変数の初出: vart
リスト要素の変数が永久変数の初出: varp
リスト要素の変数が一時変数の参照: valt

以前我々は get-list 命令と unify 系命令の複合化を検討する際、get-list 命令に続いてどのような unify 系命令が現れるかを評価した[6]。その結果によれば get-XX-YY-list 命令を導入したとしてもそれらの命令実行頻度は 30% 弱のはずであった。しかし今回の測定によれば get-XX-YY-list 命令の実行頻度は 50% を越えており命令導入の効果は大きいと言える。

5. 3 速度向上

1) "if-then-else" 最適化, "neck-cut" 最適化による速度向上

本稿で述べた最適化の効果を SIMFOS の実行性能比として測定した。測定に用いたプログラムは先に述べた静的/動的解析を行う時に使用したのと同じものである。

SIMFOS 3.2 は "if-then-else", "neck-cut" 最適化を適用せずにコンパイルされている。SIMFOS 4.2 は "if-then-else" 最適化のみを適用しコンパイルされている。そして、SIMFOS 5.0 は "if-then-else", "neck-cut" 最適化を適用してコンパイルされている。測定対象とした各 SIMFOS の版ごとに SIMFOS のソース・プログラムは多少異なる。しかし、プログラムの実行性能の違いはコンパイル時の最適化レベルが支配的であると考えられる。

表 8 SIMFOS の実行性能向上比

OSの版名	最適化レベル	実行時間	速度 向上率
SIMFOS 3.2	レベル 0	16740msec	約 6 %
SIMFOS 4.2	レベル 1	15746msec	
SIMFOS 5.0	レベル 3	15100msec	約 4 %

2) Lips 値

1) で示した実行時間を述語呼び出し回数および組込述語の呼び出し回数の和で割ることにより SIMFOS 実行時の Lips 値を求めた。逐次型の prolog 処理系の性能を現わす指標として append の Lips 値が使われるが OS レベルの仕事をしている場合のそれを求め append の場合と比較した。(PSI-II における append の Lips 値は 400 klips)。

表 9 SIMFOS 動作時の Lips 値

OSの版名	最適化レベル	Lips値	append比
SIMFOS 3.2	レベル 0	171 Klips	0.42
SIMFOS 4.2	レベル 1	198 Klips	0.49
SIMFOS 5.0	レベル 3	216 Klips	0.54

6. まとめ

- 1) PSI-II に採用した特徴的な最適化技法である "if-then-else", "neck-cut" 最適化は今までのクローズインデキシング技法ではインデキシングすることの出来なかった述語に対して高速なクローズインデキシングが可能であることを示した。
- 2) 本稿で述べた最適化技法はベンチマークプログラムのような小さなプログラムに対してばかりでなく PSI-II の OS である SIMFOS に対しても有効であり最適化によって約 9% 程度的高速化が成されたことを示した。
- 3) PSI-II に導入した複合化命令によって大規模プログラムにおいても約 1.2% 程度的高速化が成されていることを示した。また、SIMFOS のコンパイル・コードに対する静的解析によりコード量が約 2% 程度削減されていることを明らかにした。

参考文献:

- [1] D.H.D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, AI Center, SRI International, 1983.
- [2] 中島克人 他 マルチ PSI 要素プロセッサ PSI-II のアーキテクチャ 情報処理学会第 33 回全国大会 7B-3 p.149-150.
- [3] 稲村雄 他 PSI-II のシステム・アーキテクチャ 情報処理学会第 34 回全国大会 4P-9 p.161-162.
- [4] H.Nakashima and K.Nakajima, HARDWARE ARCHITECTURE OF THE SEQUENTIAL INFERENCE MACHINE: PSI-II, Proceedings of 1987 Symposium on Logic Programming, Sep, 1987, p.104-113.
- [5] 中島克人 他 PSI-II の性能評価(1) 情報処理学会第 35 回全国大会 6B-7 p.677-678.
- [6] 高木茂行 KLO 機械語への IF THEN ELSE 制御構造の導入によるプログラム実行効率の向上について ICOT IM-0104 1985.
- [7] 中島浩 他 PSI-II における KLO 実行順序系組込述語の実現方式 情報処理学会第 35 回全国大会 6B-6 p.675-676.
- [8] 立野裕和 他 PSI-II の性能評価 情報処理学会第 37 回全国大会 3Y-5 p.619-620.
- [9] 稲村雄 他 PSI-II の性能評価(2) 情報処理学会第 35 回全国大会 6B-8 p.679-680.