ICOT Technical Report: TR-445

TR-445

KL1のクローズインデキシング方式

木村　康則、西崎慎一郎、中越　靖行、
平野　喜芳(富士通SSL)、近山　隆

December, 1988

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

KL1のクローズインデキシング方式

木村康則[1]　西崎慎一郎[2]　中越靖行[2]　平野喜芳[2]
近山 隆[1]
1: (財) 新世代コンピュータ技術開発機構*
2: (株) 富士通ソーシアルサイエンスラボラトリ

概要

本論文では、並列論理型言語KL1のクローズインデキシング方式を提案し、評価結果を報告する。

KL1では、ゴールの実行のために試みる候補クローズの選択は自由に行ってよい。そこで、本論文では、コンパイル時に個々のクローズが選択されるための条件を求め、選択される可能性のあるクローズ群をまとめてコンパイルしてオブジェクトコードを生成するクローズインデキシング方式を提案する。本方式では、引数のデリファレンス、ヘッド引数として現れた構造体の分解や、組込述語などクローズ間で共通した処理は、重複して実行されないようにコンパイルされる。

本方式の効果を調べるために汎用計算機上に実装して評価した。その結果、インデキシングを行わない場合と比較して、静的なコード量は増大せず、実行時間も一割から4～5倍向上することがわかった。また、実行時の動的な命令分岐の数も減らせることがわかった。

1　はじめに

ICOTでは、第五世代コンピュータプロジェクトの一環として、並列推論マシンPIM[3]の研究開発を進めている。PIM上の言語は、並列論理型言語KL1を予定している。本論文では、KL1の高速実行を達成するための一方法である、クローズインデキシング方式について提案し、評価する。

KL1の実行では、ゴールを実行するために試みる候補クローズの選択の順番は、自由である。従って、コンパイル時に、個々のクローズが選択されるための条件を求め、選択される可能性のあるクローズ群のみを試みることにより、余分な実行を無くすことができる。また、選択される可能性のあるクローズ群をまとめてコンパイルすることにより、クローズ間で共通した処理の実行の重複を避けることができる。KL1では、Prologと異なりガード部のユニフィケーションが一方向性であることことから、比較的簡単に実現できると考えられる。

*108 東京都港区三田1-4-28 三田国際ビル21F, Tel: 03-456-3193, E-mail: ykimura@icot.jp.net

今回提案するクローズインデキシング方式では、まず個々のクローズのヘッド引数を解析し、クローズが選択されるための引数の条件を求める。次に、これを基にクローズ群を各ノードが引数の条件で終端が分類されたクローズであるようなツリーに展開する。そしてこのツリーを用いて、選択される可能性のあるクローズ群のみを試み、かつクローズ間で共通した処理の実行の重複を避けるようなオブジェクトコード列を生成する。

本論文では、まず、クローズインデキシングの基本的な考えかたを述べ、次にコンパイラでの処理方式について説明する。またインデキシングのために新たに導入された命令について説明する。さらに、汎用計算機上で行った評価結果について報告し、考察を加える。

2　KL1のクローズインデキシング

2.1　並列論理型言語KL1

並列論理型言語KL1は、フラットGHCに基づいてICOTで設計された言語である。KL1プログラムは、次のようなシンタックスを持つクローズの集合として表される。

$$H :- G_1, \cdots, G_m | B_1, \cdots, B_n. \ (m \geq 0, n \geq 0)$$

ここで、H, G_i, B_iは、各々、クローズヘッド、ガードゴール、ボディゴールと呼ばれる。オペレータ, |, は、コミットメントバーと呼ばれ、クローズ中でこれに先立つ部分を受動部(またはガード部)、これに続く部分を能動部(またはボディ部)と呼ぶ。ここで、受動部には、組込述語しか書けない。これは、GHCの言語記述能力を保ちながら、効率的な実現を考慮して採用された制限である。また、ボディのゴールには、プライオリティや実行するプロセッサを指定するためのプラグマと呼ばれる機能を付加することが出来る。

2.2　素直なコンパイル方式

KL1コンパイラは、同じ述語名、引数個数のクローズ群を一纏めにして扱い、読み込んだ順に一つ一つのクロー

```
filter(P, [X|Xs1], Ys0) :- X mod P =\= 0 |
    Ys0 = [X|Ys1], filter(P, Xs1, Ys1).      (1)
filter(P, [X|Xs1], Ys0) :- X mod P =:= 0 |
    filter(P, Xs1, Ys0).                     (2)
filter(P, [],      Ys0) :- true | Ys0 = [].  (3)
```

図 1: フィルタソースプログラム

ズを独立にコンパイルして KL1-B[1] と呼ばれる KL1 の抽象命令列を生成する。命令列の最後には、実行が中断した時 (サスペンションと呼ぶ) の処理を行う命令が置かれる。 KL1-B は、Prolog における WAM(Warren Abstract Machine)[10] に相当するものである。これが従来用いられてきた最も素直なコンパイル方式である。例えば、図 1 に示したプログラムは、図 2 のようにコンパイルされる。

この方式では、コンパイラは対象となったクローズのみに注目すれば良いため、コンパイル時間が短く、かつコンパイラの作成が簡単であると言う利点がある。一方で、クローズ間に亘った情報を考慮にいれていないため、次のような欠点がある。

- (i) 番目のクローズの選択の試みが失敗したことにより、(i+j) 番目 (j > 0) のクローズの選択が失敗すると分かってしまう場合でも、(i+j) 番目のクローズの選択の試みが行われてしまう。例えば、図 1 では、ゴールの第二引数がリスト以外のものでも、クローズ (1)、(2) の順で実行が試みられる。
- (i) 番目のクローズの選択の試みが失敗し、(i+j) 番目のクローズの選択の試みを行う時、(i) 番目のクローズの選択の試みの時に行われた両クローズに共通する処理、例えば、変数のデリファレンスや、ヘッド引数の構造体の分解など、が重複して行われてしまう。例えば、図 1 では、クローズ (1) の "X mod P = \ = 0" で失敗して、クローズ (2) の実行が行われるとき、再び第二引数のリストの分解が行われる。

そこで、同じ述語名、引数をもつクローズ群を一纏めにして扱い、各クローズが選択されるためのヘッド引数の条件およびガード部の組込述語を調べて、クローズ間に亘った情報を利用してコンパイルすることにより、このような欠点を除き、 KL1 の高速な実行を目指すことにした。

2.3 クローズインデキシングの方針

2.2 節で述べた目的を達成するために本論文で実現したクローズインデキシング方式について、基本方針をまとめる[5]。

```
filter/3:   try_me_else  filter/3/1
            wait_list  A2
            read_car  A2,  X4
            read_cdr  A2,  X5
            integer  X4
            integer  A1
            modulo  X4,  A1, X6
            put_constant  0,  X7
            not_equal  X6,  X7
            collect_list  A2
            put_list  A2
            write_car_value  A2,  X4
            write_cdr_variable  A2,  X4
            get_list_value  A2, A3
            put_value  X5,  A2
            put_value  X4,  A3
            execute  filter/3
filter/3/1: try_me_else  filter/3/2
            wait_list  A2
            read_car  A2,  X4
            read_cdr  A2,  X5
            integer  X4
            integer  A1
            modulo  X4,  A1,  X4
            put_constant  0,  X6
            equal  X4,  X6
            collect_list  A2
            put_value  X5,  A2
            execute  filter/3
filter/3/2: try_me_else  filter/3/3
            wait_constant  [],  A2
            collect_value  A1
            get_constant  [],  A3
            proceed
filter/3/3: suspend  filter/3
```

図 2: フィルタコンパイル (NoIndexing) 結果

1. ヘッドおよびガード部のユニフィケーションとガード部の組込述語をインデキシングの対象とし、各クローズが一意に決まるまでヘッドの第一引数、第二引数、... ガード部の組込述語と言う順でインデキシングの対象を拡げていく。
2. 引数のデータ型は、整数、アトム、リスト、ベクタ、ストリング、"変数であるが、ガード部の解析により、或る決まったデータタイプの値に具体化されていなければならない変数"、前記以外の通常の変数と、その他[1]、に分類する。

[1]例えば、コードへのポインタなどがある。

3. インデキシング対象の引数が、ベクタの場合には、まずその要素数で分類し、それで分類出来ない場合には、その要素を引数レジスタに読み出してきて、インデキシングの対象とする。リストの場合も同様にリスト要素をインデキシングの対象とする。

4. ヘッドの引数で分類できない場合には、ガード部の組込述語も分類の対象とし、同じ組込述語や、互いに背反であるような組込述語の実行を重複して行わないようにする。

5. 複数のクローズのガード部で以下のような同じ処理がある場合、それらをまとめ、重複した処理を行わないようにする。

   - デリファレンス及び変数かどうかのチェック
   - データタイプのチェック
   - 値一致のチェック
   - 構造体の分解
   - 組込述語

3 KL1 コンパイラの処理

KL1 コンパイラに実装したクローズインデキシングの方針について説明する。KL1 コンパイラは、まず各々のクローズが選択されるためのクローズのヘッド引数の条件を求める。次にこの条件をもとに、クローズをツリーに展開していく。その後、このツリーをトラバースしていきながらコード生成を行う。

3.1 ヘッド引数の条件の抽出

入力されたクローズ群に対して、各クローズが選択されるために必要なヘッド引数の条件を求める。以下、この条件を HCC(*H*ead arguments *C*ondition for *C*ommitment) と呼ぶ。例えば、図 1 のようなプログラムに対しては表 1 のような HCC を求める。ここで、$< d_1, d_2, \ldots, d_n >$ で一つのクローズの HCC を表し、d_i が引数位置 i の条件である。この時、ヘッド引数上は変数であるが、ガード部の組込述語の入力変数として用いられていることにより、或る決まったタイプの値が来なければ実行が中断してしまうような変数については、そのタイプも HCC に反映させる。

図 1 の例では、"var(int)" は変数が整数に具体化されていなければならないこと、"[...]" はリストに具体化されていること、"atom([])" は、アトムの [] が具体化されていること、"any" は、何でも良いことを示す。

```
(1) : < var(int), [var(int)|any], any >
(2) : < var(int), [var(int)|any], any >
(3) : < any,      atom([]),       any >
```

表 1: フィルタプログラムの HCC

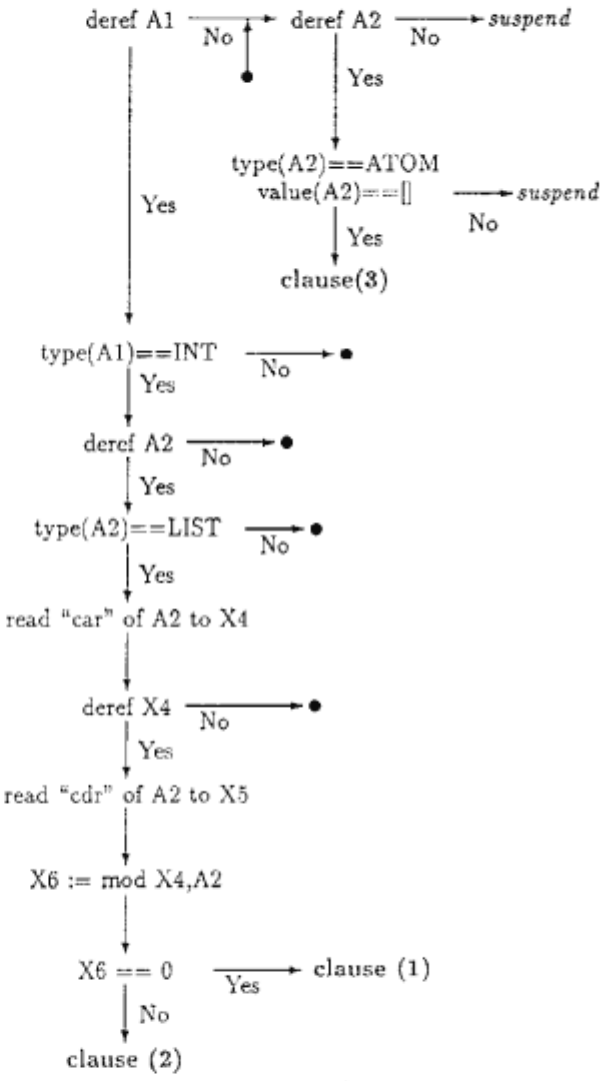

図 3: フィルタプログラムのツリー

3.2 ツリーの作成

HCC をもとに、各ノードが引数の条件による分岐、リーフ (葉) が分類されたクローズに対応するツリーを作る。この時、各ノードまでの分類で分かった引数の条件を HCC に反映させて、コンパイル時に余分な命令を生成しないようにする。図 1 の場合のツリーを図 3 に示す。

種類	命令	機能
(1)	wait Ai	デリファレンス
(2)	is_XXX Ai, Lab	タイプチェック
(3)	test_XXX C, Ai, Lab	値一致チェック
(4)	wait_XXX C, Ai	(1) + (2) + (3)
(5)	jump_on_non_XXX Ai, Lab	(1) + (2)
(6)	check_XXX C, Ai, Lab	(2) + (3)

表 2: インデキシング用の命令の種類

3.3 オブジェクトコードの生成

前節で作ったツリーをルートからたぐって行きながらコードの生成を行う。各ノードがインデキシングの命令に対応し、クローズのコンパイルでは、そこまでのツリーをたぐることによって既に実行された処理に対する命令列は生成しないようにする。

4 クローズインデキシング用の KL1-B 命令

4.1 インデキシング命令の種類

2.3 節で述べたように、インデキシングの対象となった引数に対して、次の三つの操作がツリーの各ノードに現れてくる。

- 引数のデリファレンスと具体化の確認
- 引数のタイプチェック
- 値一致のチェック

従って、これに対応する命令を、各データタイプ毎に準備した。これらの種類と機能を表 2 に示す[6]。

表 2 で、(1)、(2)、(3) が各々デリファレンス、タイプチェック、値チェックに対応する。(4)、(5)、(6) は、(1)、(2)、(3) の命令を統合した命令である (4.2 節)。ここで、"XXX" は、"list"、"vector" などの Ai の取り得るデータタイプを示す。また、引数が未定義のために実行が中断し、次候補クローズのコードへ分岐する場合と、値の不一致 (以下 "失敗" と言う) などにより、次候補クローズのコードへ分岐する場合を分け、前者の分岐アドレスは命令中の "Lab" によって指定し、後者は "try_me_else" 命令で指定するようにした。従来は、実行の中断および失敗時の分岐アドレスは全て "try_me_else" 命令によって設定されていた。このように命令に分岐アドレスを陽に持たせた理由は、インデキシングでは分岐命令が多く使われると予想され、またこの分岐先を全て "try_me_else" 命令で設定すると、"try_me_else" 命令の数が増え、かえって実行時間や静的コードサイズの増大を招くと考えたからである。

ヘッド引数に定数が現れた場合を例に、インデキシング用の KL1-B 命令について以下に説明する。

1. デリファレンス命令

   - wait Ai
     引数レジスタ Ai をデリファレンスし、未定義ならば "try_me_else" 命令 で設定された次候補アドレスへ分岐する。それ以外の場合は次命令へ進む。

2. タイプチェック命令

   - is_atom Ai, Lfail
     デリファレンス済みの Ai のデータタイプがアトムであれば次命令へ進む。 それ以外の場合は、Lfail で示される次候補アドレスへ分岐する。 デリファレンス済みの引数レジスタのデータタイプを調べる命令で、他に is_integer, is_list, is_vector, is_string がある。
   - switch_on_type Ai, Latom, Lint, Llist, Lvect, Lstr, Lfail
     デリファレンス済みの Ai のデータタイプに応じて、Latom, Lint, ... に分岐する。どのタイプでもなければ Lfail に分岐する。この命令は、 Ai の取り得るデータタイプが三種類以上ある場合のみに生成される。

3. 値チェック命令

   - test_constant Const, Ai, Lfail
     デリファレンス済みの Ai の値が Const と等しければ、次命令へ進む。 それ以外の場合は、Lfail で示される次候補アドレスへ分岐する。デリファレンス済みの引数レジスタの値の一致を調べる命令である。 同様な命令に、ベクタの引数個数を調べる test_arity 命令がある。
   - branch_on_constant Ai,
     [(C1,L1),(C2,L2),...Lfail]
     デリファレンス済みで、データタイプがアトムか整数である Ai の値 に応じて、L1, L2,... に分岐する。 C1, C2,... のどれにも一致しなければ、 Lfail に分岐する。同様な命令に、ベクタの引数個数で多方向分岐する branch_on_arity 命令がある。

4.2 命令の統合

インデキシングツリーをたぐってコードを生成する時には、各ノードでは、対応する (4.1 節で述べた) インデキシング命令を使えばよい。しかし、これらの命令は、一命令で一つの機能を実行するもので、これだけでコードを生成すると静的コードサイズが大きくなり、実行命令数も多くなる。そこで一旦命令列を生成した後に、もう一度見直して命令の統合を行うことを試みる。

命令の統合にあたっては、出来るだけ多くの命令を統合することを考え、また出来るだけ“前”の命令同士を統合することを考える。たとえば、デリファレンス、タイプチェック、値チェックの命令が並んでいる場合には、まずこの三つの命令を一つに統合することを試み、できなければデリファレンスとタイプチェック命令の統合を試みると言う順で統合を行っていく。また統合の仕方は、分岐アドレスが同じかどうかで変わる。

1. デリファレンス + タイプチェック + 値チェック命令

   引数レジスタの内容が、未定義の場合とタイプチェックや値チェックでの 失敗の場合の分岐アドレスが同じ場合の統合命令である。例として、“wait_constant Const, Ai” がある。この場合の分岐アドレスは、先立つ try_me_else 命令で設定されているので命令には分岐アドレスを持たせる 必要はない。またこの命令は、インデキシングを行わない場合に受動部の ユニフィケーションのために生成される命令に一致する。 表 2 の (4) の場合である。

2. デリファレンス + タイプチェック命令 (その 1)

   引数レジスタが、未定義の場合とタイプチェックでの失敗の場合の分岐先が 同じ場合の統合命令である。例として、“atom Ai” や、“integer Ai” などの命令がある。分岐先は、先立つ try_me_else 命令によって設定される。

3. デリファレンス + タイプチェック命令 (その 2)

   引数レジスタが、未定義の場合とタイプチェックでの失敗の場合の分岐先が 違う場合の統合命令である。未定義の場合の分岐先は、先立つ try_me_else 命令で設定され、失敗の場合のそれは、命令オペランドとして与えられる。“jump_on_non_atom Ai, Lab” や、“jump_on_non_list Ai, Lab” などの命令がある。 表 2 の (5) の場合である。

4. タイプチェック + 値チェック命令

   デリファレンス済みの引数レジスタ Ai のタイプチェックおよび値チェックを 行うための統合命令である。例として、“check_constant Const, Ai, Lfail” 命令や “check_vector Arity, Ai, Lfail” がある。 表 2 の (6) の場合である。

```
filter/3:   try_me_else  filter/3/1
            integer  A1
            wait_list  A2
            read_car  A2,  X4
            integer  X4
            read_cdr  A2,  X5
            modulo  X4,  A1,  X6
            try_me_else  filter/3/5
            put_constant  0,  X7
            not_equal  X6,  X7
            collect_list  A2
            put_list  X2
            write_car_value  X2,  X4
            write_cdr_variable  X2,  X4
            get_list_value  X2,  A3
            put_value  X5,  A2
            put_value  X4,  A3
            execute  filter/3
filter/3/5: collect_list A2
            put_value X5,  A2
            execute  filter/3
filter/3/1: try_me_else  filter/3/6
            wait_constant  [], A2
            collect_value  A1
            get_constant  [],  A3
            proceed
filter/3/6: suspend  filter/3
```

図 4: フィルタコンパイル (Indexing) 結果

4.3 コンパイル例

図 4 に、図 1 をインデキシングをかけてコンパイルした結果を示す。

5 実験および評価

5.1 実験

実験は、UNIX マシン上に構築された KL1 処理系である PDSS システム[9] を使って行った。PDSS システムでは、KL1 のソースプログラムはまず KL1-B 抽象命令にコンパイルされ、次にアセンブラによりバイトコードに変換される。そして、PDSS エミュレータのトップレベルがバイトコード列を次々に読み出し、解釈実行することにより、KL1 プログラムが実行される。

プログラム名	リダクション数	機能
prime	5876	500 までの素数
queen	38878	エイトクイーン
qlay	19419	レイヤード法による エイトクイーン
bup	34857 (ON) 35858 (OFF)	ボトムアップパーサ
klicmp	14919	KL1 で記述した KL1 コンパイラ
espascal	335115	パスカルの三角形
etsmall	918520	パズル (E.Tick)
tri	666235	トライアングル
semi	292309	セミグループ
pax	17530	並列パーサ

表 3: ベンチマークプログラム

5.2 評価項目とベンチマークプログラム

クローズインデキシングの効果を測定するために、インデキシングをかけてコンパイルした場合 (以後 "ON" と略記) と、2.2 節で述べたインデキシングをかけないでコンパイルした場合 (以後 "OFF" と略記) のふたつの場合を比較した。評価項目としては、コンパイルされたコードの静的なサイズ、実行したときの KL1-B 命令の実行数と実行時間、実行された分岐命令の総数と実際に分岐した数などである。

ベンチマークプログラムとして、表 3 に挙げたプログラムを使った[2, 8, 7]。"Bup" では三引数のマージ述語を使っており、インデキシング ON/OFF で選択されるクローズが変わるためにリダクション数が変わっている。

5.3 静的コードサイズ

表 4に、静的コードサイズを KL1-B 命令数で示す。インデキシング ON の場合の方が、サイズが小さくなっている。これは、ガード部を纏めてコンパイルすることにより、重複したコードが生成されていないためと考えられる。

5.4 実行命令数と実行時間

表 5 に実行命令数と実行時間を示す。実行時間の単位はミリ秒である。また、インデキシング OFF を基準とした場合の増減を示す。実行命令数、実行時間ともプログラムによってインデキシングの効果にばらつきが見られる。また、実行命令数の減少の割合に対して実行時間の減少の割合が小さい傾向がある。この理由としては、減った命令はガード部の実行のためのものであり、ガード部の命令

	OFF	ON	±(%)
prime	128	113	−11.7
queen	216	194	−10.2
qlay	234	200	−14.5
bup	3400	3233	−4.9
klicmp	8539	8061	−5.6
espascal	2346	2161	−7.9
etsmall	1923	1877	−2.4
tri	2484	2439	−1.8
semi	617	533	−13.6
pax	50898	45320	−11.0

表 4: 静的コードサイズ

	Indexing	命令数	±(%)	時間	±(%)
prime	ON	95267	+2.1	1690	+0.6
	OFF	93324		1680	
queen	ON	574499	−4.8	12330	−0.7
	OFF	603677		12420	
qlay	ON	433201	−34.8	8340	−30.3
	OFF	664138		11960	
bup	ON	464622	−20.1	11360	−9.8
	OFF	581555		12590	
klicmp	ON	255743	−4.9	6230	−0.2
	OFF	269056		6240	
espascal	ON	4598162	−6.0	108990	−7.3
	OFF	4889898		117520	
etsmall	ON	17250751	−22.8	667950	−13.5
	OFF	22351835		772140	
tri	ON	13751029	−4.3	322070	−3.4
	OFF	14366902		333250	
semi	ON	4570589	−28.0	217450	−7.1
	OFF	6352156		233970	
pax	ON	255014	−85.2	6910	−78.2
	OFF	1725475		31680	

表 5: 実行命令数と実行時間

は、ボディ部の命令よりも元々一つ一つは軽いと言うことが考えられる。

5.5 分岐命令数と分岐回数

表 6 に分岐命令の実行回数と実際に分岐した回数を示す。ここで、分岐命令数とは、ガード部で実行が中断あるいは失敗する可能性のある命令とゴールリダクションの切れ目などで実行される命令 ("execute"、"proceed"、"suspend"、"otherwise") の総実行回数で、分岐回数はそ

	Indexing	分岐命令数	分岐回数	±(%)
prime	ON	67380	6376	−1.5
	OFF	69403	6471	
queen	ON	417786	53768	−7.1
	OFF	438590	57880	
qlay	ON	254339	44385	−40.0
	OFF	355072	73981	
bup	ON	279317	59158	−46.8
	OFF	328108	111151	
kllcmp	ON	189938	25593	−6.4
	OFF	193320	27346	
espascal	ON	3371263	604969	−3.0
	OFF	3408224	623954	
etsmall	ON	15788827	2007708	−53.5
	OFF	18166556	4317098	
tri	ON	13132366	1236931	+0.1
	OFF	13131168	1235733	
semi	ON	3133583	586341	−57.2
	OFF	4024638	1370933	
pax	ON	176484	54557	−84.5
	OFF	779633	352741	

表 6: 分岐命令数と分岐回数

れらの命令を実行した結果、実際に分岐した回数である[2]。殆どのプログラムで、インデキシング ON の場合の方が実際の分岐回数が減っている。これは、クローズの選択にあたって、インデキシング OFF では逐次的にクローズ選択の試みを行うのに対して、インデキシング ON では、ガード部の実行をまとめて行うため、クローズ選択の失敗による分岐回数が減っているものと考えられる。この結果は、命令パイプライン機構をもったマシンにおいて、命令ストリームの乱れを減らす効果をもたらし、また、コード用のキャッシュをもったマシンにおいては、ヒット率向上に寄与するものと考えられる。従ってこれらの機能を持ったマシンでは、よりインデキシングによる性能向上が期待できる。

6 考察

本章では、5 章での実験結果のなかで比較的効果のあったレイヤードクイーン問題、並列パーサプログラムとあまり効果のなかった素数生成プログラムについて考察する[4]。

[2] "execute"、"proceed"、"suspend" では必ず分岐し、その総数はインデキシング ON/OFF に拘わらず同じである。

```
filter([[I|_]|Ins], I, K, Out) :- true | ...
filter([[I|_]|Ins], I, K, Out) :- K := I-J | ...
filter([[I|_]|Ins], I, K, Out) :- K := J-I | ...
filter([[I|In]|Ins], I, K, Out) :- I =\= J | ...
```

図 5: レイヤードクイーン (一部)

```
n_subj_zz2([],V,Out,L) :- true| Out=[].
n_subj_zz2([msg1(In,LCL)|Z],A,Out,LCU) :- ...
n_subj_zz2([msg2(In,LCL)|Z],A,Out,LCU) :- ...
n_subj_zz2([msg3(In,LCL)|Z],A,Out,LCU) :- ...
n_subj_zz2([msg4(In,LCL)|Z],A,Out,LCU) :- ...
n_subj_zz2([msg5(In,A,LCL)|Z],B,Out,LCU) :- ...
                    :
```

図 6: 並列パーサプログラム (一部)

6.1 レイヤードクイーン問題

レイヤードクイーンは、図 5 に示すように、そのプログラミングスタイルからガード部に深くネストしたリスト構造が現れている。従って、インデキシング OFF では、クローズ選択の試み時に何度もリスト構造の分解が重複して行われる。インデキシング ON では、同じ構造をしたリストの分解は一度しか行われない。この差が実行命令数や時間の差に現れている。

6.2 並列パーサ (PAX) プログラム

並列パーサ (PAX) プログラムでは、図 6 に示すように、プログラム間でのデータの受け渡しのためにリストを使ったストリームを多用する。本方式では、リストの要素もインデキシングの対象とするので、この場合には、リストの Car のベクタ構造によってインデキシングがかかり、レイヤードクイーンの場合と同様に、逐次的なリスト構造の分解と候補クローズのサーチを避けている。この場合では、"n_subj_zz2/4" と言う述語は、合計 57 のクローズから成っている。

6.3 素数生成プログラム

素数生成プログラムでは、インデキシングの効果は殆どなかった。これは、このプログラムでは、候補クローズの数が少なく、しかも、よく選択されるクローズをソースプログラム上の前の位置に書いているからである。今、クローズの順を入れ替えてあまり選択されないクローズを前に書いたプログラム ("bad_prime") [3] を実行させてみる

[3] 具体的には、図 1 のクローズ (3) を (1) の前にもってくるなどの操作を行った。

Indexing	プログラム	命令数	時間
ON	Prime	95267	1690
	Bad_prime	95267	1690
OFF	Prime	93324	1680
	Bad_prime	105488	1830

表 7: 素数生成プログラムの比較

と、表 7 のようになる。

インデキシング ON では、どちらも実行命令数、時間共同じである。インデキシング OFF では、実行命令数、時間が一割程度増えている。また、インデキシング OFF の Prime と ON の場合の実行命令数、時間の差は僅かである。このことから、インデキシングを行うことにより、プログラマは、処理系の詳細を気にせずプログラム作成を行うことができ、しかも、プログラマが最適にクローズを並べた場合に近い実行速度が得られることがわかる。

7 おわりに

クローズを一纏めにしてコンパイルし、余分なガード部での実行を行わないようにすることで、KL1 の実行を高速化することが出来ることを示した。特に、レイヤード手法を使ったプログラムに対して効果が大きいことがわかった。今後、この結果をさらに解析し、より KL1 の実行特性に合ったクローズインデキシング方式および命令セットの設計を進めていく予定である。

本方式では、インデキシング用命令の数が多くなっている。本論文での評価に用いたような汎用機上に構築した処理系では負担にならないが、専用マシン上に実現する場合には、ハードウエアなどの制限により命令数を絞る必要もでてくるであろう。ただしこの場合でも、ターゲットマシンの特性を生かすように命令セットを最適化することにより、さらに高速な KL1 処理系を実現できると考える。

謝辞

日頃御指導いただく ICOT 第 4 研究室内田俊一室長に感謝します。また、ここで述べたことの多くは ICOT の並列処理検討会での議論に基づいている。後藤厚宏 (ICOT)、中島克人 (ICOT)、関田大吾 (MRI)、宮崎敏彦 (沖電気) の各氏を始めとする同検討会メンバ諸氏に感謝します。

参考文献

[1] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *Symposium on Logic Programming '87*, pages 468–477, Aug. 1987.

[2] A. Okumura and Y. Matsumoto. Parallel Programming with Layered Streams. In *Symposium on Logic Programming '87*, pages 224–231, Aug. 1987.

[3] 後藤厚宏, 杉江 術, 服部 彰, 伊藤徳義, 内田俊一. 並列推論マシンPIM – 中期構想 –. 情報処理学会第 *39* 回 (昭和 *61* 年後期) 全国大会, 3B-5, Oct. 1986.

[4] 西崎慎一郎, 平野喜芳, 武井則雄, 森田京子, 木村康則. KL1 クローズインデキシング方式の評価. 情報処理学会第 *38* 回 (昭和 *64* 年前期) 全国大会, March 1989.

[5] 木村康則, 関田大吾, 近山 隆. KL1 におけるコード生成の最適化. 情報処理学会第 *36* 回 (昭和 *63* 年前期) 全国大会, pages 815–816, March 1988.

[6] 木村康則, 後藤厚宏, 中島克人, 近山 隆. KL1抽象命令セットの改良について. 情報処理学会第 *38* 回 (昭和 *64* 年前期) 全国大会, March 1989.

[7] 寿崎かすみ 他. マルチ PSI における並列構文解析プログラム PAX の実現および評価. 並列処理シンポジウム *'89*, Feb. 1989.

[8] E. Tick. Performance of Parallel Logic Programming Architectures. ICOT TR-421, Sept. 1988.

[9] ICOT第四研究室. PDSS - 言語仕様と使用手引き -. ICOT TM-437, 1988.

[10] D.H.D.Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.