Vol. 23 No. 1 情報処理学会論文誌 Jan. 1982

ショートノート

直接マイクロコード生成形コンパイラによるLISPマシンの高速化†

金田悠紀夫†† 前川禎男†† 瀧和男†††

高級言語マシンの研究の一環として開発したLISPマシンの性能についてはすでに報告している．本論文は，LISPプログラムを直接マイクロコードにコンパイルするコンパイラの導入により得られる効率の向上について論じている．簡単なベンチマークプログラムを用いた評価によりインタプリタに比して，5～8倍の高速性が得られることが判明し直接マイクロコード生成形コンパイラがきわめて有効であることが判明した．

1. はじめに

われわれはマイクロプログラム制御型のLISPマシンを製作しその性能についてすでに報告している[1),2)]．

本論文では直接マイクロコード生成形コンパイラ導入により期待される性能の向上について論じる．

2. LISPマシンのコンパイラ

2.1 中間言語目的コード生成形コンパイラ[3)]

現有のコンパイラは中間言語命令コード(32ビット長)を目的コードとして生成する方式をとっている．プログラム実行はマイクロプログラムで記述されたエミュレータが主メモリ上の目的コードを順次フェッチ(FETCH)し実行の制御を行っていく．中間言語命令コードは文献3)で示すように，関数呼出のための制御情報をスタック上に作成するフレーム作成命令(MKF)，関数のコール，リターン命令(CALL, RET)，スタック操作命令(PSH, POP)，分岐命令(JP)，変数のバインディング処理命令(BIND, REBIND)が用意されている．このコンパイラの働きによりインタプリタに比してプログラム実行時間が1/1.5～1/3に短縮されている．

2.2 直接マイクロコード生成形コンパイラ

中間言語目的コード実行においては，目的コードのフェッチなど実際のLISPプログラム実行に直接結びつかない操作が必要となり冗長となっている．したがってLISPプログラムを直接マイクロコード化できればより高速な処理が実現できると考えられる．ここではA，B2つのバージョンのマイクロコード生成形コンパイラを想定し若干のベンチマークプログラムのマイクロコードの生成をマニュアルで行い，LISPマシンのWCS(マイクロプログラムコード用メモリ)に格納し実行させ性能評価を行った．

LISPプログラムのマイクロプログラム化は現コンパイラによって生成された中間言語目的コードを利用し各目的コードの実行マイクロコード部分を連結して1つのマイクロプログラムにすることにより実現している．この際に行う最適化の程度によりA，Bのバージョンが想定されている．

バージョンA：中間言語命令コードのフェッチと再帰関数以外の関数呼出におけるフレーム作成の省略．

バージョンB：Aに加え以下の最適化をしている．

1. なるべく引数を内部レジスタに残しておき，スタックを介さずに引き渡すとか，スタック中の引数を取り出して再びスタック先頭に積む操作をやめ直接レジスタに読み込むなど内部レジスタの有効利用．
2. 演算結果がTかNILの条件で，リターンするか否かが決る部分では結果のT，NILをスタックに積む操作を省略し条件分岐命令JPNでリターン．
3. 並行実行可能な連続した分岐命令と演算命令は一命令に統合してマイクロコードの圧縮をする．
4. MAPCAR, MAPCONルーチンと引数関数とを直接統合することにより関数呼出やパラメータの受け渡しを簡略化する．

3. システムの性能評価

性能評価に用いたプログラムは第2回LISPコンテストにおいて出題されたプログラムからTARAI, BITA, BITB, SORTを選んだ[4)]．実行時間の測定結

† Performance Improvement of the LISP Machine by the Micro-code Object Generation Compiler by YUKIO KANEDA, SADAO MAEKAWA (Systems Engineering, Kobe University) and KAZUO TAKI (Omika Works, Hitachi Ltd.).

†† 神戸大学工学部システム工学科
††† 日立製作所大みか工場

表 1　ベンチマークプログラム実行時間 (msec)

Table 1 Execution time of the bench mark programs.

	INTERPRETER	COMPILER	VERSION-A	VERSION-B
TARAI-5	25,900	11,800	4,646	3,902
TARAI-6	952,400	429,300	170,712	143,367
BITA-7	178	80	40	31
BITA-8	618	276	136	108
BITB-8	113	73	30	23
BITB-9	377	244	101	75
SORT-80	572	209	95	75
SORT-100	749	277	123	98

表 2　TARAI-5 実行時の各ルーチンの実行ステップ数と割合

Table 2 Total executed microprogram steps and percent executed steps of each routine (TARAI-5).

COMPILER			VERSION-A			VERSION-B		
ルーチン	ステップ数	割合(%)	ルーチン	ステップ数	割合(%)	ルーチン	ステップ数	割合(%)
FETCH	9,262,959	28.3	PSH	2,229,968	16.9	PSH	1,029,216	9.0
PSH	2,573,043	7.9	MKF	1,886,900	14.3	MKF	1,972,664	17.2
MKF	7,118,753	21.7	CALL	257,304	1.9	RET	2,573,048	22.4
CALL	2,830,347	8.6	JP	686,146	5.2	EVAL	36	0.0
JP	686,146	2.1	RET	2,573,054	19.5			
RET	5,403,399	16.5	EVAL	36	0.0			
EVAL	27	0.0						
GREATERP	3,087,657	9.4	GREATERP	3,773,803	28.6	GREATERP	4,116,876	35.8
SUB 1	1,801,128	5.5	SUB 1	1,801,128	13.6	SUB 1	1,801,128	15.7
合　計	32,763,459		合　計	13,208,339		合　計	11,492,968	

```
(DE TARAI (X Y Z)
          (COND
           ((GREATERP X Y)
               (TARAI  (TARAI (SUB1 X) Y Z)
                       (TARAI (SUB1 Y) Z X)
                       (TARAI (SUB1 Z) X Y) ))
           (T Y)  ))
```

```
(LAP 3 TARAI 3
       (MKF 1 -6)
       (PSHA  -6)
       (CALL 2 GREATERP)        } 343073
       (JPN  G1)
       (MKF 3 -14)
       (CALL 2 SUB1)
       (PSHA -10)
       (PSHA -10)
       (CALL 3 TARAI)
       (MKF 2 -14)
       (CALL 2 SUB1)
       (PSHA -10)               } 85768
       (PSHA -13)
       (CALL 3 TARAI)
       (MKF 2 -14)
       (CALL 2 SUB1)
       (PSHA -13)
       (PSHA -13)
       (CALL 3 TARAI)
       (CALL 3 TARAI)
       (RET)
  G1   (RET -1)                 } 257305
```

図 1　TARAI ソースプログラムとそのコンパイルコード

Fig. 1 TARAI source program and its compiled codes.

果を表 1 に示す．各プログラムの最大実行時間のものを例にするとバージョン A，B はそれぞれ現コンパイラに比して 1/2.0～1/2.5，1/2.8～1/3.2 に，インタプリタに比し 1/3.7～1/6.1，1/5.0～1/7.7 に実行時間を短縮している．

4. プログラムの動特性の検討

プログラムの実行時間がプログラム実行ステップ数と対応がとれることを利用して各ルーチンの実行ステップ数を現コンパイラおよびバージョン A，B のコンパイラの出力コードに対して調べた．TARAI-5, BIT-8 について結果を示す．

4.1 TARAI-5 の場合（表 2）

図 1 に TARAI-5 とそのコンパイルされた中間言語目的コードが示してある．また実行したときの各部分の実行回数もあわせて示してある．

FETCH 命令の省略と命令の減少により大幅にステップ数が減少している．MKF 命令の減少は SUB 1, GREATERP の 2 つのシステム関数が再帰呼出されていないので，マイクロルーチンを直接マイクロコードに埋め込む形をとっており MKF 命令を省略してい

```
(DE BIT (A)
 (COND  ((NULL (CDR A)) A)
        ((NULL (CDDR A))
         (LIST (CONS (CAR A) (CONS '$ (CDR A)))))
       (T (BITL (CDR A) (LIST (CAR A))))))


(DE BITL (X J)
 (COND ((NULL X) NIL)
       (T (NCONC (MAPAPPEND (BIT X) (FUNCTION
       (LAMBDA (K)
        (MAPCAR (BIT J) (FUNCTION (LAMBDA (L) (LIST L '$ K)))) )))
      (BITL (CDR X) (APPEND J (LIST (CAR X)))) ))))

(DE MAPAPPEND (X FN)
  (COND ((NULL X) NIL)
        (T (NCONC (FN (CAR X) (MAPAPPEND (CDR X) FN))) ))
```

図 2 BITA のソースプログラム

Fig. 2 Source list of BITA.

表 3 BITA-8 実行時の各ルーチンの実行ステップ数と割合

Table 3 Total executed microprogram steps and percent executed steps of each routine (BITA-8).

COMPILER			VERSION-A			VERSION-B		
ルーチン	ステップ数	割合(%)	ルーチン	ステップ数	割合(%)	ルーチン	ステップ数	割合(%)
FETCH	176,360	21.6	PSH	34,700	8.9	PSH	9,190	3.1
PSH	40,488	4.9	MKF	44,410	11.1	other ARG	3,155	1.1
MKF	88,676	10.5	CALL	4,619	1.2	MKF	25,216	8.4
CALL	67,220	8.0	JP	13,141	3.3	CALL	2,596	0.9
JP	15,461	1.8	BIND	46,594	11.6	JP	9,115	3.0
BIND	69,518	8.3	RET	46,305	11.5	BIND	38,060	12.7
RET	118,324	14.1	REBIND	35,546	8.9	RET	31,252	10.5
REBIND	35,546	4.2	EVAL	28	0.0	REBIND	35,546	11.9
APPLY	30,960	3.7				EVAL	28	0.0
EVAL	21	0.0						
LIST	67,316	8.0	LIST	60,457	15.1	LIST	58,522	19.6
MAPCAR	60,694	7.2	MAPCAR	49,745	12.4	MAPCAR	33,604	11.2
CDR	15,144	1.8	CDR	15,144	3.8	CDR	12,735	4.3
APPEND	14,500	1.7	APPEND	13,775	3.4	APPEND	9,425	3.1
NCONC	14,161	1.7	NCONC	14,161	3.5	NCONC	14,161	4.7
CAR	9,072	1.1	CAR	9,072	2.3	CAR	6,663	2.2
CONS	8,448	1.0	CONS	7,250	1.8	CONS	7,040	2.4
CDDR	6,006	0.7	CDDR	5,005	1.3	CDDR	3,003	1.0
合　計	837,915		合　計	399,952		合　計	299,311	

るためである．フレーム作成を省略するとリターンの際に必要なフレームをたたむ操作も不要となり大幅なステップ数減少となる．GREATERP 関数の実行時間が，現コンパイラ，バージョンA，Bと移るにつれて増加しているが，これは結果の NIL, T をスタックに積む操作を現コンパイラでは RETURN 命令で行っていたのをバージョンAでは GREATERP に組み込み，Bではスタックに値を積まずに，T か NIL に示すフラグレジスタをテストしてジャンプする条件分岐命令 (JPN) を組み込み，スタック操作を省略したためである．

4.2 BITA-8 の場合 (図2，表3)

現コンパイラで用いている APPLY がバージョンA，Bでは省略されている．APPLY は MAPCAR で引数となっている関数を評価するのに用いられているが，バージョンAではフレーム作成と引数のスタッ

クへのプッシュ操作後にその関数にジャンプするようにし，Bでは関数を工夫しフレーム作成を省略している．Bで other ARG となっているのは引数をスタックに積まずに内部レジスタに直接入れたり，前の一連のオペレーションを行う時に読み込んだ引数をレジスタに入れて保存しておいて使うというような方法を用いて引数の評価を行うことを示している．

5. 結　論

直接マイクロコード生成形コンパイラによる効率改善について現有の中間言語目的コード生成形コンパイラと比較しながら論じてきた．バージョンBを例にとると，インタプリタに比して5～8倍，現中間言語目的コードに比較して2.5～3倍の高速性が実現されきわめて有効であることが判明した．現コンパイルコードに比して実行ステップが，

1) 目的コードフェッチルーチン (FETCH) の省略により 20%～31% (SORT-100),

2) 不要なフレーム作成の省略により 10%～25% (TARAI-6),

3) 引数の受け渡しをスタックのかわりに内部レジスタを用いて行うなどの工夫により5%～15%(BITB-9),

4) 関数呼出の工夫，BIND の際のスタック上の値の有効利用，並列演算を生かしたマイクロコードの圧縮により 7%～14% (BITA-8)

減少している．

インタプリタからバージョンBのコンパイラに移るにつれて当然のことながら組込みの SUBR 関数の実行時間比が増大しており，より高速化を目指すには高速演算回路の付加などの SUBR 関数の機能のハードウェア化が必要となってくる．

参 考 文 献

1) 瀧，金田，前川：LISP マシンの試作――アーキテクチャと LISP 言語の仕様――，情報処理学会論文誌，Vol. 20, No. 6, pp. 481-486 (1979).
2) 瀧，金田，前川：LISP マシンの試作――インタプリータの構造とシステムの評価――，情報処理学会論文誌，Vol. 20, No. 6, pp. 487-493 (1979).
3) 金田，小林，前川，瀧：コンパイラ導入による試作 LISP マシンの効率改善について，情報処理学会論文誌，Vol. 22, No. 2, pp. 114-120 (1981).
4) 竹内：第2回 LISP コンテスト，情報処理，Vol. 20, No. 3, pp. 192-199 (1979).

(昭和56年4月7日受付)
(昭和56年6月16日採録)