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†

金田悠紀夫†† 小林康博††
前川禎男†† 瀧和男†††

われわれは LISP プログラムの高速実行を目的とした LISP マシンの開発を行いインタプリタの高性能性を示したが，本論文ではコンパイラの設計とその性能に焦点を合わせて報告する．本システムはマイクロプログラム制御であるためいわゆる機械語命令は持っていない．ここでは機械語命令に対応した中間言語命令を設定し，それを目的コードとして出力するコンパイラを作成した．中間言語命令は基本的には7種類で，関数呼び出しのための call return 処理命令やスタック操作命令が中心となっている．コンパイルされたプログラムの LISP マシン上での実行時間を第2回 LISP コンテストに出題されたプログラムを用いて測定したが，超大形機上での LISP コンパイラの出力コードの実行時間にほぼ匹敵しその高性能性が実証された．またインタプリタによる実行時間とコンパイルコードの実行時間比はほぼ 1：1.5～1：3 程度であり，ほかのシステムに見られるような大幅な改善は望めないことが判明した．

1. まえがき

われわれは LISP 言語で記述されたプログラムの高速実行を実現するために LISP マシンの開発に取組んできた．試作された LISP マシンとインタプリタの性能についてはすでに報告している[1),2)]．本論文では新たに開発したコンパイラの機能と性能に焦点を合わせて報告する．

2. LISP マシンのコンパイラ

2.1 プログラム実行の制御構造

本システムでは EVAL を始めとする全関数ルーチンは入出力処理の一部を除きすべてマイクロプログラム化されている．マイクロプログラムレベルでの再帰呼び出しやハードウェアスタックが有効に利用できるようにプログラム実行の制御構造を次のように構成した．関数を実行する場合スタック上にフレームが作られる．フレームには関数本体を示すポインタ，マイクロプログラムの戻り番地を示す old μ-PC，プログラム部をたどる PC レジスタの値を格納する old-PC 1つ前のフレームを指す old FP のほか実引数値や計算の中間結果も格納される．関数の評価が終るとスタックの先頭を関数の評価値として FP の指す領域に置き現フレームを消滅させ，戻り番地へ間接ジャンプして戻る．この制御構造のもとで関数の実行を行う．

2.2 中間言語命令

本システムはマイクロプログラムにより制御されているため，機械語命令を本質的には持たない．そのためコンパイラを構成する場合機械語命令に対応した中間言語命令を設定し，それをオブジェクトコードとして生成する．コンパイルされた関数の実行時にはマイクロプログラムで記述されたエミュレータ（命令のフェッチと実行の制御をする）により中間言語命令が順にフェッチされ実行される．

これら中間言語命令の設定の際，メモリ参照があまり負担にならないこと，命令語長を1語 (32 ビット) にするなどに留意して命令コードを構成した．

（1） CALL 命令：オペランドで示された関数を，その属性に応じた実行の予備操作（引数のバインディングとかリターンの前備準）を行い，関数を呼出す．

（2） PSH 命令：オペランドで示されたデータをスタックの先頭に積む命令であり，オペランドとしては(ⅰ)実行する関数の実引数，(ⅱ)現在の自由変数値(ⅲ)データそのもの，の3種類指定でき以下の命令も同様．

（3） POP, STORE 命令：現在のスタックの内容をオペランドで示されるフィールドにコピーする命令で SET, SETQ に対応している．POP はコピー後スタックをポップアップする．

（4） MKF 命令：関数の実行に先だちフレームヘッドを作る命令で，PSH 命令をかねることも可能．

† Performance Improvement of the Experimental LISP Machine by the Compiler by YUKIO KANEDA, YASUHIRO KOBAYASHI, 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
††† (株)日立製作所大みか工場

（5） JMP 命令：条件によりラベル先へ分岐するか否か制御する命令でありスタックをポップアップすることも可能である．条件として（i）無条件分岐，（ii）NIL 分岐，（iii）T分岐の3種類が指定できる．

（6） RET 命令：現在のスタック先頭値を関数の評価値として呼び出し元へリターンする．あわせてフレームの消滅処理を行う．

（7） BIND 命令：コンパイル時に宣言した自由変数が束縛を受ける際にバインド処理をする．

以上7種の命令はオペランドの指定により26個の中間言語命令となる．

2.3 最適化

本システムのコンパイラにおける最適化処理は，生成された中間言語オブジェクトコードの圧縮という形で行っている．これは実行時における命令コードのフェッチとデコードによるオーバヘッドを軽減させるもので以下に示すことが行われている．

（1） 複数個のフレームを作るときは可能なかぎり1つ MKF 命令を用い個数はオペランドで指定する．MKF 命令の直後の PSH 命令はオペランドで指定．

(MKF 1 NIL)(MKF 1 NIL)(PSHA-5)→(MKF 2-5)

（2） RET 命令直前の命令が PSH 命令であった場合は RET 命令のオペランドで PSH 命令を指定する．T や NIL を評価値として RETURN する場合，TNIL 専用の RET 命令を用いる．

(PSHA-4)(RET)→(RET-4)

(PSHQ T)(RET)→(RET T)

（3） COND の条件クローズに NULL，NOT が用いられたときはそれらの関数は呼出さずに引数によりその判断を行う．

（4） ジャンプ命令が2つ並び前のものが条件ジャンプである場合，条件ジャンプの条件を変えることにより無条件ジャンプが削除できる場合には削除する．

…(JPN GO) (JMP B1) GO…→…(JPT B1)…

現時点では以上のような最適化を行っているだけであるが，より高度な最適化については現在検討している．

3. システムの性能評価

第2回 LISP コンテストに出題されたプログラムを用いて実行時間を測定した結果を**表1**に示す．なお表記法は文献4）にしたがっている．インタプリタIは文献2）で示した数字であり，インタプリタIIはその後改良して高速化したインタプリタである．コンパイラはコンパイルされたコードの実行時間で，中間言語命令コードのフェッチとデコードをエミュレータが行っているためこのオーバヘッド時間が含まれていることもあり，1.5〜3倍程度の速度向上になっている．

4. TARAI-3を用いたインタプリタおよびコンパイラの機能分析

本 LISP マシンにおいてコンパイラがインタプリタに対していかなる形で高速化をもたらしたかを分析するため，最も簡単でかつインタプリタの動的機能をよく利用している TARAI-3 を例に上げて分析を試みる．図3に TARAI-3 とそのコンパイルコードを示してある．TARAI-3 は

(TARAI　6　3　0)

を実行することになる．なお TARAI-3 を実行したときのプログラムの各部分の実行回数もあわせて図3に示してある．この実行回数のデータを基にして以後の分析を進めている．

インタプリタIIの流れ図を**図1**に，コンパイルコード実行をするエミュレータの流れ図を**図2**に示す．〈〉内に示された数は各ブロック内で実行されるマイクロプログラムステップ数を示したものである*．

図3は TARAI-3 をインタプリタで実行するときの実行の流れと各ブロックの実行回数，各ブロックの処理において実行されたマイクロプログラムステップ数を示している．各ルーチン群の実行ステップの内訳を示すと以下のようになる．

eval および eval ①	12.6%
apply ②	20.1
argeval	25.9
cond	9.6

表1 テストプログラム実行時間 (msec)

Table 1 Execution time of the test programs.

プログラム名	インタプリタI	インタプリタII	コンパイラ
SORT-20	73	68	25
SORT-60	415	387	135
SORT-100	804	750	262
TARAI-3	55	52	23
TARAI-4	1,013	954	429
TARAI-5	27,538	25,938	11,663
TARAI-6	1,011,732	952,988	428,487
TPU-4	1,815	1,674	960
TPU-5	238	222	106
TPU-6	6,728	6,237	3,105

* ステップ数は代表値でありすべての場合について必ずこのステップ数であるとは限らない．また流れ図は TARAI-3 を実行するのに必要な部分のみを示しており，ほかの部分は省略してある．

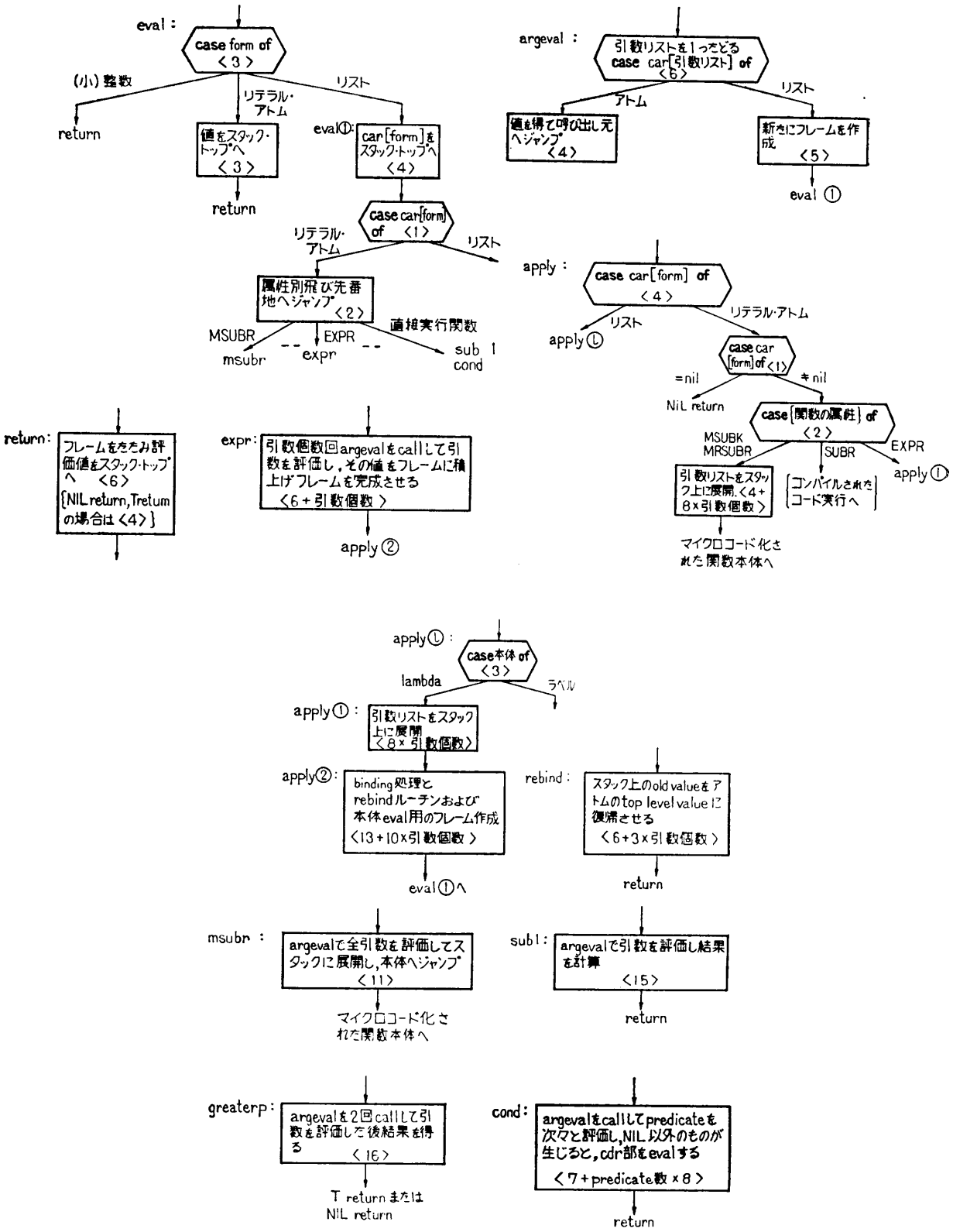

図 1 インタプリタの流れ図

Fig. 1 The flow chart of the interpreter.

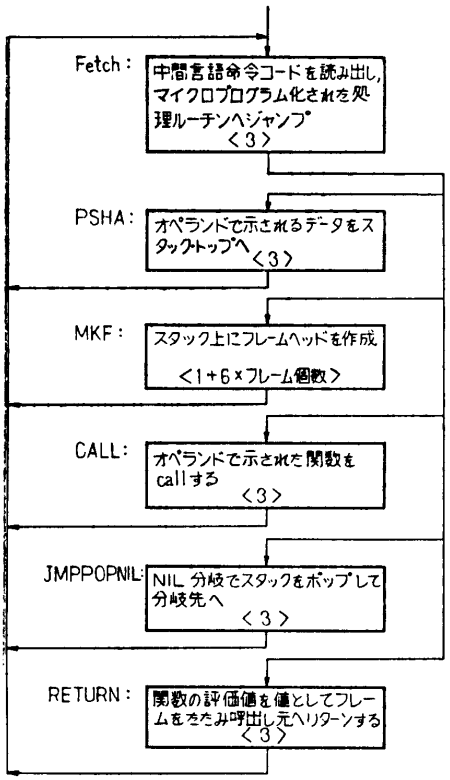

図 2 コンパイルコード実行の流れ図

Fig. 2 The flow chart of execution of compiled codes.

return	7.9%
rebind	7.2
expr	1.0
sub 1	7.5
greaterp	7.6

またコンパイルコードを実行したときのマイクロプログラムステップ数を図3にコンパイルコードとともに示してある．各ルーチンの内分けを示すと以下のようになっている．

Fetch	26.1%
PSHA	9.5
MKF	16.8
CALL（直接実行関数）	4.8
CALL（コンパイルされた関数）	2.8
JMPPOPNIL	2.7
RETURN	2.7
SUB 1	15.5
GREATERP	19.4

本例において**図4**のインタプリタの流れと図3のコンパイルコード実行の流れとはよく対応

```
(DE TARAI (X Y Z)                                    } 673 回
 (COND
  ((GREATERP X Y)
   (TARAI (TARAI (SUB1 X) Y Z)                       } 168 回
          (TARAI (SUB1 Y) Z X)
          (TARAI (SUB1 Z) X Y)))                     } 505 回
   (T Y) ))

   (LAP SUBR TARAI 3
       (MKF  1 -6)                                   } 673 回
       (PSHA   -6)                                     〈32034 ステップ〉
       (CALL 2 GREATERP)
       (JMPPOPNIL GO)
       (MKF  3 -14)                                  }
       (CALL 2 SUB1)                                 }
       (PSHA   -10)                                  }
       (PSHA   -10)                                  }
       (CALL 3 TARAI)                                }
       (MKF  2 -14)                                  }
       (CALL 2 SUB1)                                 }
       (PSHA   -10)                                  } 168 回
       (PSHA   -13)                                  } 〈34776 ステップ〉
       (CALL 3 TARAI)                                }
       (MKF  2 -14)                                  }
       (CALL 2 SUB1)                                 }
       (PSHA   -13)                                  }
       (PSHA   -13)                                  }
       (CALL 3 TARAI)                                }
       (CALL 3 TARAI)                                }
       (RET   )                                      }
  GO   (PSHA   -2)                                   } 505 回
       (RET   )                                      } 〈6060 ステップ〉
```

図 3 TARAI とそのコンパイルコード

Fig. 3 TARAI and its compiled codes.

がとれていることが判る．両者の実行時間に大きな差が出る最大の理由は apply② および rebind→return で表わされているバインディングのための処理で本例ではインタプリタの実行時間中 31.3% を占めている．コンパイルされればローカル変数はスタック上にとられるのでバインディングは不要となり最も大きな効率

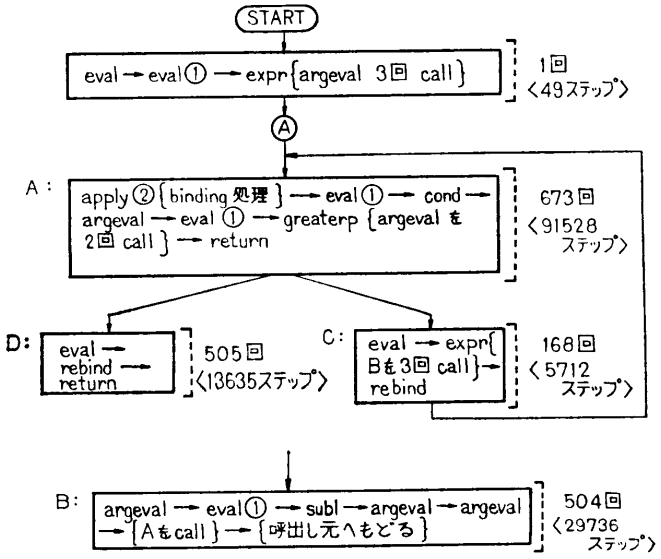

図 4 EVAL (TARAI 6 3 0) の実行フロー

Fig. 4 Execution flow of EVAL (TARAI 6 3 0).

改善の要因となる．それに続くものとしてはインタプリタでは COND の処理に 9.6% のステップを要しているが，コンパイルされればオープン展開されるためこのステップがほぼなくなる．また argeval に対応する処理としては PSHA と MKF があるが実行ステップ数では前者に比してほぼ半分程度のステップとなっている．しかしながら PSHA 命令，MKF 命令のフェッチ時間を含めると argeval の実行時間と同等になり argeval ルーチンの高速化がインタプリタの高速化に寄与していることが判る．図1のインタプリタの流れ図からも実行時の判定処理の部分が多いことが判るがいずれも少ないステップ数で実現されており本マシンの強力な条件分岐機能が有効にはたらいていることが判る．コンパイルコードの場合，フェッチ部分が占めるステップ数が大きいがこの問題については6章で論じる．

5. TPU-6 を用いた場合の動特性

次にプログラムサイズの最も大きな測定用プログラムである TPU を用いて行ったインタプリタIIとコンパイルコードの性能の測定結果について論じる．

実行時の総ステップ数と各ルーチンの総ステップ数を求めた結果が**表2**と**表3**である．また各ハードウェアオペレーションの利用回数を比率で集計したものを**表4**に示す．ここでいう比率は各ハードウェアオペレーションを含むステップ数が総ステップ数に占める割合のことである．これらのうちでメモリオペレーションとジャンプオペレーションについては，各オペレーション数の合計を100に集計しなおし，その内訳を算出したものを**表5**，**表6**に示す．表4と表6より条件ジャンプの使用は 40.1% と高くまたその内

表 2 TPU-6 実行時間とマイクロプログラムの実行ステップ数

Table 2 Execution time and microprogram steps of TPU-6

	実行時間 (msec)	マイクロステップ数	平均実行時間
インタプリタII	6,237	18,621,434	335
コンパイラ	3,105	9,714,310	320

表 3 TPU-6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 execution time of routines (TPU-6).

インタプリタII				コンパイラ			
	ルーチン	ステップ数	割合 (%)		ルーチン	ステップ数	割合 (%)
システム関数	EVAL	1,818,316	9.8	エミュレーション部	FETCH	1,345,009	13.8
	ARGEVL	2,526,304	13.6		PSHA	480,633	4.9
	PROG	1,886,867	10.1		MKF-A	309,643	3.2
	その他	2,112,731	11.4		その他	878,929	9.0
SUBR関数	SUBST	3,363,520	18.1	SUBR関数	SUBST	3,293,744	33.9
	EQUAL	2,371,520	12.7		EQUAL	2,195,884	22.6
	CAR	570,245	3.1		CAR	131,595	1.4
	NULL	338,776	2.1		NULL	525	0.0
	CDR	291,395	1.6		CDR	67,245	0.7
	その他	3,001,725	17.7		その他	1,011,143	10.4

表 4 TPU-6 におけるハードウェア機能利用率 (%)

Table 4 Utilization of hardware facilities (TPU 6).

オペレーション		インタプリタII		コンパイラ	
メモリアクセス	read	11.9	10.7	11.2	10.4
	write		1.3		0.8
ジャンプ	条件ジャンプ	40.1	15.7	36.3	13.4
	その他のジャンプ		24.4		22.9
スタックオペレーション			51.2		59.5
直接数値演算			19.9		20.2
ALU-オペレーション			21.3		16.4
FEX-オペレーション			0.6		0.6
MAP-データ使用			4.1		0.03
IDLE			0.4		0

* 総ステップ数中に含める各オペレーションを含むステップ数の割合

表 5 TPU-6におけるメモリオペレーションの内訳(%)

Table 5 Classification of memory operations.

メモリオペレーション内訳(%)	インタプリタII	コンパイラ
read car or cdr	53.2	26.8
read car and cdr	36.2	65.9
write	10.6	7.3
read while write	7.0	0

表 6 TPU-6 における直接ジャンプの内訳 (%)

Table 6 classification of the direct jump ops.

オペレーション比 (%)	インタプリタII	コンパイラ
無条件ジャンプ	47.3	46.9
条件ジャンプ	52.7	53.1
jump with other op.	87.1	98.8

表 7 中間言語命令の先取りをした場合の推定実行時間と改善率

Table 7 Improvement of execution time by a prefetching.

	実行時間	推定実行時間	改善率 (%)
TARAI-6	428,487	315,989	26.2
SORT-100	262	175	33.3
TPU-6	3,105	2,625	15.5

87.1% が ALU 演算と並列処理されていることが判る．この並列処理は総実行ステップの 34.9% にあたり並列処理が本システムにおいて大きく高速処理に貢献していることが判る．インタプリタIでは並列処理を行うジャンプ命令の全ジャンプ命令に対する比率は 70.6% であった[2]のに対し，レジスタファイルの割当，Am 2910 のレジスタ／カウンタの使用等により改良を行ったものである．

メモリオペレーションは表4よりメモリアクセスが総ステップ数の 11.9% もあるにもかかわらず純粋のメモリ待ちの比率（表中の IDLE）が 0.4% と小さくここでもメモリアクセスと ALU まわりの並列演算が行われていることが判る．また表5においてインタプリタにおける read while write オペレーションはバインディング処理のためのものである．

次にコンパイラによる場合を考えてみる．まず表3より，インタプリタにおけるシステム関数部の実行ステップ数が約823万ステップであったのに対し，コンパイルした場合のエミュレーション部の実行ステップ数が約300万と 63.4% も短縮されていることが判る．また CAR, CDR 等のマイクロプログラム化された直接実行関数では引数評価ルーチンの call および RETURN 処理の省略により大幅に実行時間が短縮されている．また NULL に関しては **2.3** で述べたように関数を呼び出さないで引数のみで NULL 機能を行うためステップ数が大幅に減少している．またコンパイルした場合のジャンプ命令の内訳をみた場合，並列演算を行うジャンプ命令の比率が 98.8% と非常に高いことが判る．表5に示されているようにコンパイラにおいては read car and cdr が多いが，これはコンパイルコードのフェッチにおいてこのタイプの read が実行されるためである．

6. 中間言語命令先取りの効果

表1の実行時間でインタプリタとコンパイラの場合を比較してみると，コンパイルすることによっても 1.5〜3 倍程度の処理速度の向上しか得られていない．これはインタプリタが高速に動作しているためでもあるが，中間言語命令のフェッチおよびデコードをマイクロプログラムで実現しているためと考えられる．特に命令コードのフェッチを行う部分は関数の実行とは直接関係のない部分である．そこで，このフェッチによるオーバヘッドをなくすために，ハードウェアによる命令コードの先取りを行った場合の実行時間を推定してみた．つまり，プログラム領域の命令コード専用の PC（プログラムカウンタ）を用意し，あらかじめ命令コードを読み出しておき，その命令コードの実行が始まると，PC により次の命令コードを読み出すという機能をハードウェアで実現した場合の実行時間である．**表7** はこのようなハードウェアを設けることによりフェッチのオーバヘッドが完全に取り除かれた場合の推定実行時間である．表7より TARAI, SORT のような組み込み関数をあまり使用しないプログラムでは 25〜30% の改善率が期待できるが，TPU のように組込関数を多用するプログラムでは 15% 程度の改善率となっている．また分岐命令によるパイプライン処理の損失による改善率の低下も考えられるが，マイクロ命令中の分岐命令実行頻度は TARAI-3 で 1.9%，TPU-6 で 1.6% なのでこれによる影響はさほどないと考えられる．

7. むすび

本 LISP マシンにおいてコンパイラがどのようにプログラム実行の高速化に寄与したかを中心に報告した．本コンパイラは LISP ソースプログラムを中間言語命令コードからなるオブジェクトプログラムに変換しエミュレータにより実行を行わせるものであり，測定の結果インタプリタに比して実行時間を 1/1.5〜1/3 に短縮することができることが判明した．

コンパイラの導入による効果はシステムによって大幅に異なるが一般に数〜数十倍の速度向上をもたらす例が多く本システムにおいてはその効果がほかに比して小さいことが判る．これは汎用計算機上のインタプリタで大きな問題となる実行時の判定処理がハードウェア機能を強化することできわめて高速化されていることが大きいと考えられる．したがって分析のところでも述べたが TARAI-3 の場合はインタプリタの大きなオーバヘッドは変数のバインディング処理と COND 処理だけとなり，ほかの部分はコンパイルコード実行と大差ないところまでインタプリタを高速化できた．今後の問題点としてはコンパイルコードのフェッチ機構のハードウェア化と，ソフトウェアでは中間言語命令コードの形ではなく直接マイクロコードのオブジェクトに LISP ソースプログラムを変換するコンパイラの開発とその効果についての測定が上げられる．

なお，本研究に協力して下さった三上昭弘，高岸英之の両君に感謝します．

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) Taki, K., Kaneda, Y. and Maekawa, S.: The Experimental LISP Machine, IJCAI proceedings, pp. 865-867 (Aug. 1979).
4) 竹内郁雄：第2回LISPコンテスト，情報処理，Vol. 20, No. 3, pp. 192-199 (1979).
5) 山口，島田：仮想計算機によるLISPプログラムの動特性，電子通信学会論文誌，Vol. J 61-D, No. 8, pp. 517-524 (1978).
6) Teitleman, W.: INTERLISP Reference Manual, Xerox, (Feb. 1974).
7) 金田，小林，前川他：試作LISPマシンの高速化について，電子通信学会電子計算機研究会，EC 79-58, pp. 71-79 (Jan. 1980).

(昭和55年4月7日受付)
(昭和55年9月18日採録)