記号処理 64－4
（1992. 3. 16）

Tachyon Common Lisp コンパイラの高速化方式

新谷 義弘* 伊藤 丹二* 長坂 篤*

*沖電気工業(株) 総合システム研究所

CLtL2 準拠の高速かつ高い移植性を持つ Common Lisp 処理系である Tachyon Common Lisp の実現方式について述べる。本処理系は現在 i860^{TM} RISC CPU を搭載した UNIX ワークステーション OKI Station 7300 上で動作する。Tachyon Common Lisp は実行速度を重視するため、従来から行なわれている Tail Recursion の繰り返しへの変換などの Lisp 固有の最適化の他に、RISC CPU 特有の最適化を含む各種の高速化を行った。Tachyon Common Lisp は、現在最も広く使われている商用 Common Lisp 処理系に対して、SPECmark でハードウエア性能を正規化した値で、インタプリタが 3.06 倍、コンパイラが 6.02 倍高速であり、現在最も高速な Common Lisp 処理系である。本論文は、Tachyon Common Lisp のコンパイラにおける、高速化手法及び RISC プロセッサ i860^{TM}特有のコンパイル手法について述べる

Optimization Techniques in Tachyon Common Lisp Compiler

Yoshihiro Shintani* Tanji Ito* Atsushi Nagasaka*

*Oki Electric Industry Co.,Ltd. Systems Laboratories
11-22, Shibaura 4-Chome, Minato-ku, Tokyo 108, Japan

The optimization techniques adopted in the compiler of Tachyon Common Lisp . Tachyon Common Lisp is an efficient and portable implementaion of the second edition of Common Lisp, are reported. The system is currently implemented on a UNIX workstation OKIstation 7300 with a i860^{TM} RISC CPU. For efficiency, Tachyon takes some optimization techniques including both Lisp level optimization such as tail recursion trnasformation and techniques developed for RISC CPU such as instruction scheduling. This peper describes optimization techniques adopted in Tachyon Common Lisp Compiler.

1 はじめに

我々は、Common Lisp 第2版 [1]に基づいた処理系 Tachyon Common Lisp を、RISC プロセッサ i860™を CPU として持つ UNIX ワークステーション OKIstation7300 上に 開発した [2]。
この処理系のコンパイラは、

1. 高速なオブジェクトコード
   高速に実行できるコードを生成するために、Peephole 最適化等の一般的な最適化の他に Lisp 言語独特の最適化を行う。
2. コンパイラの移植性が高い
   特定の CPU に依存しない、すなわち、他の CPU への移植が容易にできる。

を特徴として持つ。
コンパイラの概要については、すでに報告済み[3]であるので、本論文では、開発した Common Lisp 処理系のコンパイラにおける、

1. 高速なオブジェクトコードを生成するため用いた最適化技法
2. RISC プロセッサ i860 特有の変換技法

について述べる。

2 コンパイラの構造

まず、コンパイラの構造を簡単に述べる。
コンパイラは、以下のフェーズよりなっている。

1. 前処理フェーズ
   ・マクロ展開
   ・flet や labels のローカル関数等のインライン展開化
2. 変数解析及び文法エラーチェックフェーズ
   ・変数解析
   変数の型を決定、定数の畳み込み
   ・文法エラーの検査
3. 上位中間言語変換フェーズ
   プリミティブな関数へ変換
   ・クロージャ,セル消費関数の削除
   ・map 系関数の展開
4. 中間言語変換フェーズ
   特定の CPU に依存しない中間言語に変換
   ・実行木解析
5. アセンブリ言語変換フェーズ
   特定の CPU に対するアセンブリ言語に変換
   ・インライン展開
   ・実行木解析
6. 機械語変換フェーズ
   特定の CPU に対する機械語に変換
7. ロードフェーズ
   メモリ中への書き出し

2.1 中間言語仕様

中間言語は、約 40 個のプリミティブで Lisp プログラムの動作を表現する。以下に、仕様の概略を述べる。

2.1.1 スタック

Lisp 用にフレームやデータを積むためのスタックを 1 本用意する。これは、仮想的なもので、現実のマシンのスタックとは異なる。尚、本処理系には、スタックが、Lisp 用と システム用の 2 本あるが、システム用は、中間言語で操作しないため考慮する必要がない。

2.1.2 レジスタ

中間言語で使用する特殊目的レジスタを規定するものである。

1. 移植性を高めるために、CPU のレジスタ数を指定できる(i860 の場合 32)。
2. レジスタの用途分類
   レジスタを何に使うかを指定するもので、例えば、以下の様なレジスタの定義がある。
   - 引数
   - 戻り値
   - 引数の数
   - 戻り値の数
   - nil
   - スタックポインタ
   - 関数実体レジスタ

2.1.3 プリミティブ

中間言語のプリミティブは、関数インタフェース、引数検査・操作、多値の操作、関数の生成、データ転送、スタック操作、分岐、動的分岐、エラー、ベクタ操作よりなる(表1)。

表1：中間言語の各プリミティブ

プリミティブ	内容
entry	関数エントリ
subroutine	関数、ラムダ式のエントリ
closure	クロージャのエントリ
end	関数の終わり
return	関数から戻る
call-to-system	システム関数のコール
go-to-system	システム関数への jump
call-to-global	関数のコール
go-to-global	関数への jump
call-to-local	ローカル関数のコール
go-to-local	ローカル関数への jump
check-arg-num	引数個数検査
parse-arguments	引数の解析
nth-value	多値の値の取りだし
make-function	関数実体を作成
make-closure	関数実体を作成
move	値の移動
set-frame	フレームを作成
reset-frame	フレームを削除
label	ラベルの宣言
jump	無条件分岐
jump-t	値1 /= nil で分岐
jump-nil	値1 = nil で分岐
jump-eq	値1 = 値2 で分岐
jump-neq	値1 /= 値2 で分岐
jump->	値1 > 値2 で分岐
jump-<	値1 < 値2 で分岐
jump->=	値1 >= 値2 で分岐
jump-<=	値1 <= 値2 で分岐
case	多分岐
case-number	分岐する数字が連続している場合
set-catcher	キャッチャの設定
reset-catcher	キャッチャの解除
throw	スロー
set-protect-form	unwind-protect の設定
reset-protect-form	unwind-protect の解除
error	RUNTIME-ERROR を呼出す
get-from-vector	値をベクタより取りだす
set-to-vector	値をベクタに設定

これらのプリミティブに特徴的なものは、「属性コード」と名づけられた第1引数を持つことである。この属性コードは、特に関数呼出しのプリミティブに有益である。例えば、

```
(go-to-system ((tail . t) (arg1 . integer))
              a-function 1 )
```

は、関数 a-function への jump で、引数が1個で型が integer であることを表している。

属性コードには、

1. 引数の型
2. テイル命令である
3. 再帰命令である
4. 削除不可命令
5. 現在のスタックの深さ
6. スタック操作
7. 最適化情報(safety 、space 、speed)

等がある。これらは、アセンブラ変換時やインライン展開する場合に有益な情報となる。

3 最適化技法

プログラムを単純に機械語に変換しただけでは、冗長度が大きく、また、不必要な処理を行なってしまう。それを除くために、最適化を行ない、高速化、コードサイズの縮小をはかる。

本コンパイラでは、様々な最適化を行っているが、そのうちの代表的なものについて述べる。

3.1 最適化宣言子

declare 等の宣言における optimize 指定子に使用できるものは、CLtL2 では、

1. speed
2. space
3. safety
4. compilation-speed
5. debug

の5つである。

このうち、Tachyon Common Lisp では、speed 、space 、safety の3つの quality の値によりコンパイルコードを生成する。

尚、値は、

1. 0
2. 1
3. 2 or 3

の3段階になっている。既定値は、1である。

1. speed
   < などの述語において、safety の値が高い場合には、型宣言した変数についても型が number かどうか等を検査しているが、speed の値が高い場合には、宣言された変数については省くことができる。
2. space
   インライン展開を行なうか、行わないかを値

によって決定している。

例えば、関数 car は、値が 2 、3 の場合、インライン展開しない。

また、関数 characterp 等の述語の多くは、値が 0 の場合のみインライン展開を行う。

3. safety

   0 の場合、インライン展開時、エラーチェック部分を省略し、エラーチェックなしの専用ルーチンを使用する。

3.2 コンパイラ専用関数

コンパイルされたコードが呼び出す実行時関数として、以下のような関数を用意した。これらは、高速化やメモリの消費をおさえることを目的とする。

1. 列関数などにおける型専用関数
   merge-list など 26 関数
2. 列関数などにおける特定キー専用関数
   assoc-eq など 8 関数
3. レスト引数を持つ関数における固定引数関数
   append の 2 引数版等 81 関数
4. キー引数を持つ関数における必須引数関数
   string=6 など 24 関数
5. エラー検査のない関数
   nchk-car など 57 関数

3.3 変数の型解析による最適な実行関数への変換

Lisp は、型宣言をしなくてもよい言語である。例えば、算術演算関数の多くは、整数や浮動小数点数等の 6 つのデータ型を引数として使用できるが、コンパイラがそのままその関数を呼び出すのはインタプリタと処理速度が変わらない。そこで、宣言されていない引数についてプログラムを解析し引数の型を推論し、最適な専用関数へと変換する。また、変数等の型が確定することで、型検査などを省いたコードに変換することもできる。本コンパイラでは、関数中で宣言された変数だけでなく。

1. システム関数の引数の型
2. システム関数の戻り値の型

の情報を活用する。ここで、システム関数とは、CLtL2 にある関数すべてと本言語処理系の内部関数のことを言う。例えば、

(setq *len* (length *x*))

であれば、この式以降は、*len* は負でない fixnum 型、*x* は列型であることが推論できる。本コンパイラは、CLtL2 にある関数すべてと本言語処理系が提供するライブラリ関数についての引数と戻り値の型をデータベース化して持っている。

この最適化は、以下の処理で効果がある。

1. 引数の型がわかった場合、その型の専用関数へ変換する。

```
(delete item list)
        ↓
(delete-list item list)
```

2. 引数の型がわかった場合、型のエラー検査を省いたコードへ変換できる。

```
第1引数が simple-vector であるか?
第2引数が fixnum であるか?
(svref simple-vector index)
        ↓
(svref simple-vector index)
```

   また、戻り値の型がわかるということは、個数もわかるので、

3. 戻り値の個数がわかった場合、多値の処理が簡単にできる。

   `(mutiple-value-setq (x y z) (floor a b))`
   の場合、

```
(floor a b)                    (floor a b)
check 引数> 1                  x = 戻り値1
x = 戻り値1 or nil             y = 戻り値2
check 引数> 2           →      z = nil
y = 戻り値2 or nil
check 引数> 3
z = 戻り値3 or nil
```

3.4 ダイレクトコール

ダイレクトコールとは関数をアドレスで直接呼出す方法である。

Lisp の一つの特徴にダイナミックリンクがあり、それは、ある関数で呼出している関数が再定義さ

れると、呼出し時には、自動的に新しい関数定義を実行するようにする機構である。つまり、呼び出す関数は、呼出しをかけた時点で決定される。ところが、これは、コンパイルする時点(正確には、コードをメモリに割りつける時点)で呼出し関数のアドレスを決定してしまうダイレクトコールにとって大きな障害となっている。

本コンパイラでは、ダイレクトコールの情報を保持し、再定義時に呼出しアドレスを書き換えることでダイレクトコールを実現する。ダイレクトコールを使用した場合の関数呼出しは、簡単に示すと図１、図２のようになる。

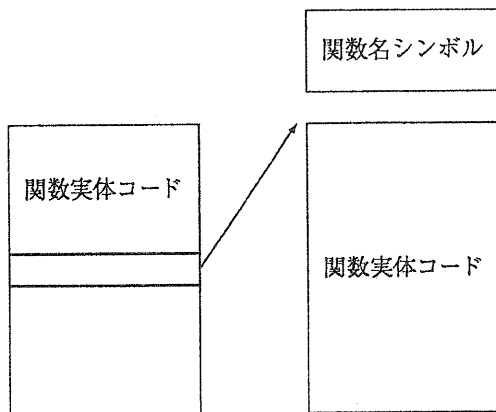

図 1: 呼出し関数がコンパイルされている場合のダイレクトコール

ダイレクトコールに関する情報は、呼出し側と呼出される側の両方に保持している。
まず、呼出す側は、相手の関数の開始アドレスがコードにあるわけであるから、それを書き換えるための情報として、以下のようなものを持つ。

1. アドレスがあるコード位置オフセット

2. 命令種類(call 、branch 等)

3. 間接呼出しのためのコード位置オフセット

次に、呼出される側であるが、これは、呼出し元の

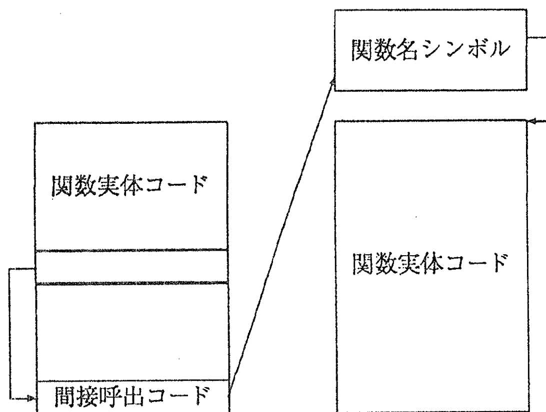

図 2: 呼出し関数がコンパイルされていない場合のダイレクトコール

関数名(または、関数実体)をリストで持っているだけである。

3.4.1 キー引数を持つ関数へのダイレクトコール

列関連の関数など、キー引数を持つ関数へのダイレクトコールは、呼び出し側でキー引数を処理し、キー引数を処理した後へ飛び込む形式で行う。

キー引数をコンパイル時に解析することで、実行時の高速化をはかっている(図３)。

3.5 実行命令制御

実行命令の並びかえ等によりパイプラインの効率化を図る。本コンパイラでは、遅延スロットを最小にするように最適化を行っている。i860 の遅延は、3 種類ある。

1. 遅延分岐
   call 、br 等の分岐命令は、1 命令余分に実行してから、制御を飛び先番地に移す。例えば、

```
addu    %r16 , %r16 , %r16
br      to-label
addu    %r16 , %r16 , %r16
```

   という命令は、実際には、

```
addu    %r16 , %r16 , %r16
```

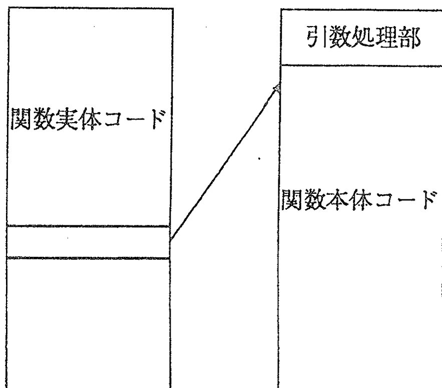

図 3: キー引数を持つ関数へのダイレクトコール

```
addu    %r16 , %r16 , %r16
br      to-label
```

というように実行される。上記例のように、実行に差し支えない命令を分岐前後より埋め込む

2. 遅延演算分岐
   演算命令の直後の条件分岐は、遅延が起きる。例えば、

   ```
   mov  %r8,   %r15
   subs %r16,  %r17, %r0
   bc   label
   ```

   と言う命令があった場合、2番目の命令の実行結果のフラグを3番目の命令でを使用しているため、遅延が起きる。

   ```
   subs %r16,  %r17, r0
   mov  %r8,   %r15
   bc   label
   ```

   とすれば、この遅延がなくなる。このように、実行に差し支えない命令を分岐前後より埋め込む
3. 遅延ロード演算
   ロード命令のディストネーションレジスタをその直後の演算命令のソースオペランドにした場合に遅延が起きる。例えば、

   ```
   ld   -6(%r16) , %r31
   addu %r31,  %r30, %r30
   ld   -2(%r16) , %r16
   ```

   と言う命令があった場合、1番目でロードした値をすぐ2番目の命令で使用しているため、遅延が起きる。

   ```
   ld   -6(%r16) , %r31
   ld   -2(%r16) , %r16
   addu %r31,  %r30, %r30
   ```

   とすれば、この遅延がなくなる。このように、実行に差し支えない命令を前後より移動させる

4 i860 特有の変換技法

本節では、i860 特有の変換技法について述べる。

4.1 アドレス指定

i860 チップには、分岐命令に 2 種類ある。

1. 相対ジャンプ(call , br 等)
   26 ビットで指定(実際には、28 ビットに相当)
2. レジスタ指定の絶対ジャンプ(calli , bri)
   32 ビットで指定

である。i860 は、即値が、16 ビットしか与えられないので 32 ビットのアドレスを生成するために、2 命令必要になる。

```
orh {上位 16 ビット} , %r0 , %r31
or  {下位 16 ビット} , %r31 , %r31
```

である。残念ながら、相対ジャンプでは、すべてのアドレスを指定できないため、システム関数以外(つまり、ユーザ定義関数)へのダイレクトコールは、絶対ジャンプを使用している。そのため、分岐には、3 命令を要する。また、ある特定アドレスの内容をロードしたりセーブするような命令も 3 命令を要する。

4.2 即値

前節でも述べたが、即値は、16 ビットの制限がある(5 ビットの制限がある特殊分岐命令もある)。即値においても 16 ビット以上のものは、

```
orh {上位 16 ビット} , %r0 , %r31
or  {下位 16 ビット} , %r31 , %r31
```

のように 2 命令を要する。

4.3 fixnum のオーバーフロー

Tachyon Common Lisp では、タグを工夫したことで fixnum を通常のレジスタ加減算と同じように扱える。i860 チップには、整数演算の結果オーバーフローがあると割り込みを起こす命令があり、fixnum から bignum への変換は、この割り込みが起きたときにすればよく、起きない場合は fixnum だとして処理できる。このため演算後の余分な検査が省かれる。

例えば、fixnum 同士の足し算では以下のコードで十分である。

```
adds %r16 , %r17 , %r16
intovr
```

5 性能評価

5.1 型推論

ここでは、型推論がどのくらいの効果があるかを gabriel のベンチマーク[4]の関数 Tak で、型宣言がない場合と、引数の型宣言があり、型が分かる場合とで比較する。従って、関数 tak は、

```
(defun tak (x y z)
  (declare (type fixnum x y z))
  (if (not (< y x))
      z
      (tak (tak (1- x) y z)
           (tak (1- y) z x)
           (tak (1- z) x y) )))
```

となるわけであるが、

関数名	回数
<	63609
1−	47706

という回数分だけ数値の比較や演算が行われる。ここで、引数をすべて fixnum だとして見ると、数値として比較や演算を行っていた、関数が、型検査が必要でなくなり、fixnum 専用の型検査なしルーチンを呼出せる。本コンパイラの場合、fixnum 検査は、2 命令で行っているので、

```
(63609 * 2 + 47706) * 2 = 349848
```

となり約 35 万命令の節約になる。
関数 tak の場合は、引数の型宣言を行うことで、

	型推論なし	型推論あり
tak	0 秒 061	0 秒 031

となり 2 倍近い速度差となる。

5.2 ダイレクトコール

ダイレクトコールがどのくらいの効果があるか見てみる。今度は、gabriel のベンチマークのうち Boyer を例にする。Boyer は、14 個の関数から成っているが、

関数名	呼び出し回数
one-way-unify1	171145
rewrite-args	169804
rewrite-with-lemmas	152280
one-way-unify1-lst	100601
rewrite	91024
one-way-unify	73499
apply-subst-lst	11448
truep	207
falsep	150
tautologyp	111
tautp	1

合計 779782 回の関数呼出しがある。
シンボル経由で呼出す場合、2 命令を使用して絶対アドレスを生成しダイレクトコールをする場合、そして、相対アドレスでダイレクトコールをする場合の 3 つの方法で計測すると、

	シンボル経由	絶対アドレス	相対アドレス
Boyer	1 秒 650	1 秒 310	0 秒 830

となり、シンボル経由と比較したダイレクトコールの高速性が実証された。

5.3 gabriel のベンチマーク

現在の gabriel のベンチマークの実行速度を表 2 に記す。

表2 ベンチマーク関数の実行速度

	インタプリタ	コンパイラ
Tak	1秒589	0秒031
Stak	3秒031	0秒249
Ctak	2秒317	0秒145
Takl	12秒671	0秒382
Takr	1秒918	0秒085
Boyer	26秒600	0秒830
Browse	39秒910	0秒480
Destructive	7秒350	0秒188
Init-traverse	53秒700	1秒220
Run-Traverse	267秒400	4秒310
Derivative	4秒090	0秒320
Data-Driven derivative	4秒940	0秒400
Division by 2 iterate	5秒480	0秒180
Division by 2 recursive	4秒520	0秒170
Fft	2秒170	0秒160
Puzzle	40秒340	1秒510
Triangle	480秒370	17秒110

処理系自体の性能評価を行なうために SPECmark でハードウェア性能を正規化して比較した結果、Tachyon Common Lisp は、現在最も広く使われている商用 Common Lisp 処理系[5] と比較して、インタプリタで 6.02 倍、コンパイラで 3.06 倍（いずれも、幾何平均）高速であり、現在最も高速な Common Lisp 処理系である。

6 おわりに

現在、マシンコードレベルの最適化は、当初予定した全ての最適化を行なっていず、実行命令制御など一部に限られている。今後、アセンブラレベルの最適化を強化し、中間語レベルを含めたその他の最適化の研究、評価をおこなう。

現在、最適化によるコンパイル時間の増大があるが、これは、最適化においてセルメモリを大量に消費する等アルゴリズムに問題がある。これらの改良も行う予定である。

また、i860 以外のプロセッサへの移植を行なう予定である。

参考文献

[1] Guy L. Steele Jr.:"Common Lisp the Langauge Second Edition" Digital Press, 1990

[2] 大江他 :"OKI Common Lisp の開発 –概要–" 情報処理学会第４３回全国大会、５Ｌ－２

[3] 新谷他 :"OKI Common Lisp の開発 –コンパイラ–" 情報処理学会第４３回全国大会、５Ｌ－４

[4] Richard P. Gabrile :"Performance and Evaluation of Lisp Systems" The MIT Press,1985

[5] "Lucid Common Lisp Reference Guide" Lucid,Inc,1986