記 号 処 理 55－3
（1990．6．4）

プロファイラデータを利用した動的最適化Ｌｉｓｐコンパイラ

岩井輝男　中西正和
慶應義塾大学 計算機科学科 中西研究室

コンパイラを実行させた場合に各基本ブロックが実行されるには多くの場合偏りがある。なぜならば一般にIF文のTHEN部とELSE部が実行される頻度は異なっており、またループの内側は外側より多く実行されるからである。ところが、コンパイラは実行の頻度の偏りを正確に知ることはできなくそのため十分な最適化が行なえない。このためプロファイラというツールを用いて人間の手で基本ブロック実行頻度を解析してプログラムを最適化をしてきた。この過程を組み込んだ最適化を行なうコンパイラを作成した。この最適化法を動的最適化と呼ぶことにする。

Lisp compiler of Dynamic Optimization with profile data

Teruo Iwai , Masakazu Nakanishi
Depertment of Conmputer Science , Keio University

Execution frequencies of basic blocks of a computer program are not equal in most cases because execution frequencies of THEN-part and ELSE-part of IF-statement is different and because code inside loop is executed more frequently than outside. Current compilers cannot optimize programs sufficiently because execution frequencies of basic blocks are not known at compile time. Usually, program are tuned further by hand analyzing execution frequencies of basic blocks with tool called Profiler.

We implemented a Lisp compiler that performs the tuning process above for optimizing object code. We named this optimization technique Dynamic Optimization.

1　序論

現在はプログラムの実行時間の短縮にはコンパイラを使っている。また最適化を行うコンパイラを使うことでそれ以上にスピードが上がる。またコンパイラの最適化されたコード以上にスピードを上げるには現在、プロファイラで実行データをとり、それを解析して人間の手でプログラムを最適化している。プロファイラとはプログラムを実行している時にその実行データを採り、そのデータを解析するプログラムチューンナップツールである。しかし現在のプログラムは巨大化し人間の手で最適化を行うには大変なことである。

このプロファイラの実行データを採りそのデータを解析してその結果を使ってプログラムの最適化行う方法を動的最適化と呼ぶことにする。

この動的最適化はプログラムの大きい範囲についての最適化ではなく、局所的な範囲での最適化である。多くのプログラムは複数の小さなサブルーチンによって構成されているが、実行時間の中でプロシージャの呼び出しに関するオーバーヘッドは決して小さいものではない。例えば引き数と結果の引き数渡しの処理がある。このような理由から、実行のデータを集めその統計を使ってそのプログラムの関数呼び出しの同じ部分での呼びだし回数が解るので、その回数が大きいときはその関数が小さいものであれば、その関数を展開できる。また、プロファイラデータで条件分岐命令の条件を満たす割合が解ることから、IF式やCASE式の分岐命令の実行回数を減らすことができる。

ここではLisp言語においてのこのような方法のコード生成の新しい考えの動的最適化について述べた。

2 動的最適化

このシステムはKCL[7]上で作成した。この動的最適化Lispコンパイラは「ソースプログラムの最適化を行い、Lispのソースを出力する」である、ソースコードコンパイラである。出力されたLispのプログラムはKCL上のコンパイラでコンパイルを行う。また、KCL上で作成したので、その他の最適化はKCLのコンパイラが行っている。KCLの行っている最適化はピープホール最適化、テイルリカージョンである。この動的最適化コンパイラにおいての最適化の方法は以下の通りである。

（1）ピープホール最適化

IF式の条件式を変えることでTHENの部分とELSEの部分を換えることにより、分岐命令をある程度実行しないようにする。CASE式はIF式に変換されるがこの時、評価順序を替えることで、評価する部分を減らすことが可能である。

（2）コードが小さく、評価回数の多い再帰ではない関数を展開

関数が多いと関数呼び出しによるオーバーヘッドは小さくはない。このため、よく呼ばれる関数などで、コードの小さな関数を関数呼び出しにするのではなく、呼び出した側の関数の内部に展開する。

プロファイラを使い、そのデータをとることによって、上で述べた最適化を行うことが可能である。以下に、2種類の最適化の理論、及び効率を述べる。

2．1　プロファイラのデータ

この最適化コンパイラのプロファイラで解るデータは以下の通りである。

（1）関数の評価にかかった時間

（２）関数の呼ばれた回数

（３）IF式とCASE式での部分的な実行回数

一般のプロファイラは（１）と（２）しかデータを取ることができないのだが、ここで使うプロファイラでは（３）のデータを取ることが可能である。

この３つのデータを採ることの可能なプロファイラを使用することで最適化が可能である。

２．２ LispのIF式の条件式の変更

LispのIF式の条件式の部分を変更することでTHENの部分とELSEの部分を取り替えることが可能である。プロファイラを使ってTHEN部とELSE部のどちらの方を評価する方が多いかを調べることによって分岐命令をあまり実行しないで済むようになる。その例と理由を以下に述べる。

```
(defun foo (x y)
  (if (zerop x)
      (setq y (1+ y))  ;foo-if-1-then
      (setq y (1- y))) ;foo-if-1-else
  (print y))
```

プログラム2.2.1　サンプルプログラム１

表2.2.1　サンプルプログラム例の
プロファイラの結果
単位［回］

Block Name	calls
foo-if-1-then	1950
foo-if-1-else	50

プログラム2.2.1は関数fooの引き数xが０の時引き数ｙをインクリメントして、そうでない時yをデクリメントし、yの値をを出力するプログラムである。この関数fooが他の関数から評価されているものとする。その時にプロファイラによって表2.2.1という結果が得られたものと仮定する。表2.2.1のBlock Nameとはプロファイラが IF式のTHENの部分とELSEの部分につけた名前である。その各部分を実行した回数がcallsの値である。表からIF式のTHENの部分を評価する回数が多いという結果である。プログラム2.2.1をコンパイルした時の機械語のコードをプログラム2.2.2に示す。

```
 1:  foo:                              ;実行回数［回］
 2:       tst    x                     ;  ２０００
 3:       bne    if-else ;分岐命令１   ;  ２０００
 4:  if-then:
 5:       inc    y                     ;  １９５０
 6:       br     if-end  ;分岐命令２   ;  １９５０
 7:  if-else:
 8:       dec    y                     ;      ５０
 9:  if-end:
10:       jsr    pc,print              ;  ２０００
11:       rts    pc                    ;  ２０００
```

プログラム2.2.2　サンプルプログラム１のコード

プログラム2.2.1のIF式の内部のTHENの部分はプログラム2.2.2の5,6行目であり、ELSEの部分は8行目である。プログラム2.2.2に各行の数は実行回数である。

プロファイラデータからIF式の内部で8行目よりも5,6行目を実行する回数が多い。プログラム2.2.2でIF式の内部では、5と6行目は1950回実行され8行目は50回実行されている。この6行目の分岐命令の実行を少なくするには分岐命令の条件を変えることによってTHENの部分とELSEの部分を入れ換えることで可能である。

条件を変えてTHENの部分とELSEの部分を入れ替えたプログラムをプログラム2.2.2に挙げる。

```
 1:  foo:                              ; 実行回数［回］
 2:       tst    x                     ;   ２０００
 3:       beq    if-else   ;分岐命令１;   ２０００
 4:  if-then:
 5:       dec    y                     ;      ５０
 6:       br     if-end    ;分岐命令２;      ５０
 7:  if-else:
 8:       inc    y                     ;   １９５０
 9:  if-end:
10:       jsr    pc,print              ;   ２０００
11:       rts    pc                    ;   ２０００
```

プログラム2.2.3　サンプルプログラム１の最適化したコード

```
(defun foo (x y)
  (if (not (zerop x))
      (setq y (1- y))
      (setq y (1+ y)))
  (print y))
```

プログラム2.2.4
サンプルプログラム１の
最適化されたプログラム

このように最適化を行うとプロファイラの結果からIF式の内部の5と6行目は50回の実行であり8行目は1950回の実行である。これを実行すると分岐命令２は50回で済む。このことからプログラム2.2.2とプログラム2.2.3の実行時間の差は分岐命令２の1900回実行の差となる。これをLispのプログラムに戻すとプログラム2.2.4となる。プログラム2.2.1をプログラム2.2.4のように変換することで実行時間を減らすことが可能であることが解る。

ただし、この場合は条件式のELSEの部分に、評価される割合の高い式をいれたがすべての場合においてこのようにはならない。例えばTHENとELSEの部分に各々にｒｅｔｕｒｎがある場合、テールリカージョンする場合、ｇｏｔｏ式で分岐する場合などは当てはまらない。何故ならばこの例の分岐命令２に当たる分岐命令がなく、ここに他の関数への分岐命令、またはリターン命令などが入るためである。

２．３　メモリスペースの小さい関数の展開

メモリスペースの小さな関数の展開できる例として以下にプログラム2.3.1を挙げる。

```
(defun main()
  (setq s 0)
  (dotimes (i 100)
      (setq s (+ s (func i)))))
(defun func (x)
  (1+ x))
```

プログラム2.3.1　サンプルプログラム２

```
(defun main()
  (setq s 0)
  (dotimes (i 100)
     (setq s (+ s (1+ i)))))
(defun func (x)
  (1+ x))
```

プログラム2.3.2
サンプルプログラム２を
最適化したプログラム

Lispのプログラム2.3.1の上で、func関数は呼び出されてそしてまた“1+”関数を呼び出す。そして、計算が終わるとfunc関数に戻りmain関数に戻る。これは無駄なのことである。何故ならばfunc関数を呼ばないで、main関数から直接“1+”関数を呼び出す方が実行時間が速くなる。この理由として、関数呼び出しを行うと引き数の受け渡しのプロセスを行ってから実行に移るからである。しかもfunc関数はこの場合100回評価されるので無駄な引き数受け渡しのプロセスを100回行っていることになる。よってこの関数をmain関数の中に展開することで実行時間の短縮となる。

関数の展開を行うとプログラム2.3.2となる。ただしfuncという関数は削除しないで残しておかなければならない。何故ならばLispはインタープリタであり、この関数が他に使われるかどうか判断できないために削除できない。

２．４　CASE式の評価順序の変更

CASE式の評価順序変更の説明をプログラム2.3.3に示す。

```
(defun bar (x)
  (case x
    (1 (* x 2))
    (2 (+ x 1))
    (3 (/ x 2))
    (4 (* x x))
    (t (print 'error))))
```

プログラム2.3.3
サンプルプログラム３

```
(defun bar (x)
  (if (eql x 1)                ;Block Name
      (* x 2)                  ;if-1-then
      (if (eql x 2)            ;if-1-else
          (+ x 1)              ;if-2-then
          (if (eql x 3)        ;if-2-else
              (/ x 2)          ;if-3-then
              (if (eql x 4)    ;if-3-else
                  (* x x)      ;if-4-then
                  (print 'error))))))
                               ;if-4-else
```

プログラム2.3.4
サンプルプログラム３を展開したプログラム

このbar関数はCASE式の評価値を戻り値とする。KCLのCASE式はIF式に展開されてプログラム2.3.4になる。この関数が他の関数から評価されている場合にプロファイラで実行データを取りその結果が表2.3.1と仮定する。

表2.3.1
サンプルプログラム３の
プロファイラデータ例
単位［回］

Block Name	call
if-1-then	10
if-1-else	190
if-2-then	10
if-2-else	180
if-3-then	175
if-3-else	5
if-4-then	4
if-4-else	1

```
(defun bar (x)
  (case x
    (3 (/ x 2))    ;if-1
                   ; この条件を満たす割合が最も高い
    (1 (* x 2))    ;if-2
    (2 (+ x 1))    ;if-3
    (4 (* x x))    ;if-4
                   ; この条件を満たす割合は最も低い
    (t (print 'error))))
                   ;else
```

プログラム2.3.5
サンプルプログラム３を
最適化したプログラム

この結果によるとIF式の３番目の部分まで評価される割合が高い。そこでIF式の条件文の順番を換えることで多くの条件式を評価しないで済むようになる。しかもIF式の順を換えても結果は変わらない。よってプログラム2.3.4をプログラム2.3.5に最適化する。プログラム2.3.5のように、CASE式の条件を満たす割合の高い節を先にいれるようにした方がよい。何故ならば、最初の条件を満たさなければ2番目の条件を評価し、それでも満たさなければ次の条件を評価するので、できるだけ早くに条件を満たすようにした方が無駄な条件評価を行わなくて済むからである。

この最適化を行う場合と行わない場合を比べる。このプログラムで条件文の順番の変更を行った時に、プログラム2.3.5にあるブロック名のある部分まで実行したときにそれまで条件式のIF式の比較を行った回数を数えたのが、表2.3.1である。

表2.3.1　サンプルプログラム３の最適化前後の比較　単位［回］

Block	ｉｆ－１	ｉｆ－２	ｉｆ－３	ｉｆ－４	全体
最適化前	10*1	10*2	175*3	5*4	565
最適化後	175*1	10*2	10*3	5*4	245
前後差					320

これから解るようにこの最適化を行うことで条件式をの比較を

５６５－２４５　＝　３２０回

より320回行わないで済むようになる。これによって実行のスピードが速くなる。

3　結果及び結論

この動的最適化を使ったLispコンパイラを使って実験プログラムの最適化を行い、実行時間を計測したところ実行時間の短くなるプログラムと短くならないプログラムがあることがわかった。

この最適化法の欠点はプログラムに偏りがないときには最適化の効率が上がらない。しかし、一般のプログラムの場合プログラムの偏りがあるのでこの最適化法は効率の良く、実用に十分に耐えうるコンパイラの最適化であることが解る。また、この動的最適化法は他の最適化法にはできないことであり、これらを組み合わせることで十分に効率の良い最適化となる。

またこのプロファイラを使って変数のレジスタ割付けを行うことも可能である。ここではLisp言語での最適化を行ったがプロファイラデータを用いた動的最適化法は他の言語でも応用が可能である。

4　参考文献

[1] Wulf William, et al.
The Design of an Optimizing Compiler,
Elsevier New York, 1975

[2] Michael Karr
Code Generation by Coagulation
Preceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction

[3] Vivek Sarkar, IBM Research
Determining Average Program Execution Times and their Variance
Proceeding of the ACM SIGPLAN '89 Symposium on Compiler Construction

[4] Susan L. Graham, Peter B. Kessler, Marshall K. McKusick
gprof: a Call Graph Execution Profiler
Preceeding of the ACM SIGPLAN '82 Symposium on Compiler Construction

[5] Unix Programmer's Manual,"prof command"
Bell Laboratories
[6] R.M.McCLURE
Object Code Optimization
Communications of the ACM Vol12 No 1. Jan 1969
[7] Taiichi Yuasa,Masami Hagiya
Kyoto Common Lisp Report
Teikoku Insatu INC
[8] SunOS System Call Manual
Sun micro systems