初期のコンパイラの開発と最適化

中田育男
筑波大学電子・情報工学系

東京大学大型計算機センタの初代の計算機HITAC 5020のFORTRANコンパイラをどのように考えて設計したか，その結果はどうであったかを述べ，そこで考えたこと，特に最適化について，その後の歴史を述べる．

Early optimizing FORTRAN compiler for the first computer of the Computer Center of University of Tokyo is described. How it was designed, the evaluation of the design, and the brief history of the optimizations are also described.

1．はじめに

筆者は昭和35年（1960年）に日立製作所の中央研究所に入社し，そこで計算機というものを初めて見たが，当時は「自動プログラミングの研究」という名のもとでアセンブラの開発が行われていた．筆者の最初の仕事はHITAC502という制御用計算機のアセンブラ，次がHIPAC103というパラメトロン計算機のアセンブラで，その次がコンパイラであった．最初に開発したコンパイラは，HIPAC103のFORTRAN II コンパイラで，これは高橋延匡，吉村一馬の両氏と共同で開発した．この時はとにかく作るということに精一杯であった．次に開発したのがHITAC5020のFORTRAN IVコンパイラ（HARPと名づけた）である．この時は2回目の開発であって，コンパイラが一応わかっていたから，単に作るのでなく，良いものを作るということに意を用いることができた．設計らしい設計をしたのはこの時であるので，この時の経験を主に書くことにする．このコンパイラが東大大型計算機センターの初代の計算機のコンパイラとして稼動を始めたのは昭和40年である．ここでは主として，何に重点を置いて，どのようにしてそのコンパイラを開発したかを述べ，最後に，そのようなコンパイラの技術の後日談またはそれらがその後どのように発展したかに簡単に触れてみたい．

2．HITAC 5020 FORTRANの開発

IBMが最初に作った704用のFORTRANコンパイラ[2][3]が作り出す目的コードの性能は今考えても素晴らしいものであった．アセンブラでプログラムを作ってきた人達にFORTRANを使ってもらうためには，アセンブラで書くのと同じ位効率のよいプログラムになることを示さねばならぬと，FORTRANの開発者は考えて，そのために心血をそそいだからである．その結果が素晴らしく効率のよい目的コードとなったのであるが，

その代償としてコンパイル速度の低下をまねいていた．アセンブラに代ってFORTRANになればデバッグは不要になると考えた[3]のもそのように設計した理由であろうと思われるが，我々はFORTRANレベル（ソース・レベル）でのデバッグがしやすいものにすべきだと考えた．そのために，(1) コンパイル速度を速くし，ソース・レベルでの修正をしやすくすること，(2) デバッグ文を提供すること，(3) コンパイル時のエラーチェック機能を高めること，(4) 実行時のエラーに対してはその発生までの経路をエラーメッセージとして出力すること，などを実現することにした．

我々が基本設計の段階で最も意を用いたのは，コンパイル速度が速く，しかも効率のよい目的コードを生成する，という目標をどのようにして達成するかであった．FORTRANで目的コードの効率をよくするためにはDOループの部分の効率をよくすることが必要である．ループの中によくあるのは配列要素の参照である．それには番地計算が必要である．たとえば，

```
DIMENSION A(10, 20)
.....
B＝A(I, J)
```

の場合，A(I, J)の番地は，A(s, t)の番地を$A_{s,t}$と書くことにすれば，

$$A_{1,1}+(I-1)+10\times(J-1) \qquad (1)$$

または，

$$A_{0,1}+I+10\times(J-1) \qquad (2)$$

として求められる（ただし各要素は1語を占め，番地は語単位についているとする．当時はバイトという言葉はなかった）．ループの中にA(I, J)があった時，このような番地計算を毎回しないですませる方法を考えるのが設計のキーポイントである．(1)式や(2)式の中の加算は機械語の命令の番地修飾の機能をできるだけ使うことにするとして，問題は掛算である．これはあらかじめ図1のような掛算の結果を並べた表を目的プログラムの中に作っておいて実行時には掛算をする代りにその表を参照することにした．

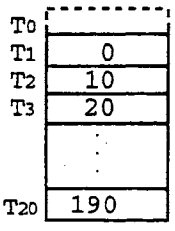

図1　掛算の表

この表とHITAC5020にあったMNI命令（Modify Next Instruction：次の命令の番地を修飾する命令）を使うと，A(I, J)の値をレジスタA8に加える命令は次の2命令ですむ．

```
MNI  0, T0#j                    (3)
AF   A8, A0,1#i                 (4)
```

ただし，j, iはレジスタ番号でレジスタjにはJの値，レジスタiにはIの値が入っていると

する．命令(3)でT_0＋J番地の値，すなわち10×(J－1)の値がとり出され，それが次の命令の番地部の値に加えられる．次の命令の番地部の値は$A_{0,1}$＋Iであるから結局$A_{0,1}$＋I＋10×(J－1)番地の内容がレジスタA8に加え（AFはAdd Floating命令）られることになる．あとは，実行回数が多いと思われるループ（内側のループ）の中でA(I, J)が使われていたとき，IとJの値がレジスタに割り当てられるようにすればよい．それは，あまり大変でなくできそうである．3次元の配列についても同様に2命令でやる方法を考えることができる．以上のことで，704 FORTRANコンパイラがやっているような膨大な設計/製作作業をすることもなく，コンパイル時に大きな時間をかけることもなく，結構効率のよい目的プログラムが得られる見通しがついた．

目的コードの設計でさらに特色を出そうと思ったのは，アキュムレータ（累算器）として使われるレジスタの使い方である．HITAC 5020はそれまでの計算機と違って15個のレジスタを持っていた．そのうちの7個はインデックス・レジスタとしての機能を持っていたので，上記のIやJの値や番地計算のために使うことにし，残りの8個を算術式の計算などのアキュムレータとして使うことにした．この複数個のアキュムレータを生かした最適のコードを生成するアルゴリズムを考えて実装した[5].

目的コードでもう一つ工夫したのは，論理式の目的コードである．論理式が，たとえば，

A .OR. B

の形をしている時，Aの値（真か偽）を計算し，さらにBの値を計算し，その2つの値のORをとるのでなく，Aが真なら，Bの値のいかんにかかわらず全体の値は真であるから，Bの計算はしないようにすることが考えられる．そのためには「Aが真ならBの計算を飛び越す」コードを生成すればよい．論理式の構文解析をしながらそのようなコードを生成するためのアルゴリズムを書いた論文は当時すでにあったが，それは「Aが真ならどこへ飛び，偽ならどこへ飛ぶ」というコードが作られるものであった．しかし，上の例でもわかるように，2つの飛び越し命令のうち，どちらか1つは次の命令に飛ぶものであり無駄な命令である．この無駄な命令は別のパスで取り除くことも考えられるが，筆者は初めからそのような無駄な命令を作らないアルゴリズムを考えたいと思った．そしてそのアルゴリズムができたと思って，それを実装に組み込んだ．（実はそれには誤りがあった．）

幾つかの細かい失敗がなかったわけではないが，この開発プロジェクトは，全体としては，開発期間についても，出来上がったものの品質・性能についても，成功であったと思う．そうなった要因には，後になって考えてみれば，次のようなことがあると思う．

（1）2回目の開発

1回目とは規模も目標も大きく違いはしたが，コンパイラの開発は2回目であったので，全体を見通して設計を進めることが出来た．

（2）チーフ・プログラマ・チーム

ストラクチャード・プログラミングが提唱された時，開発体制はチーフ・プログラマ・チームがよいといわれたが，本開発はまさにその体制で行われた．チーフ・プログラマ

が全体のフェーズの設計をし，それが出来た段階で，各フェーズをサブ・チームが担当していった．各サブ・チームのリーダはベテランで優秀なプログラマであった．

（3）デザイン・ノートブック

設計に当たって考えたことはすべて文書にまとめていった．この文書は，手書きA4版で二，三百ページのものになったが，これはHARPコンパイラの設計思想または設計方針を書いたデザイン・ノートブックに当たる．このプロジェクトの貧弱な文書体系の中にあって，これは，途中からプロジェクトに参加した人にとっても，後の保守担当者にとっても有用な文書であったと思う．

（4）余計なマネージメントなし

このコンパイラは，激烈な受注戦争の結果，東大の大型計算機センタの初代の計算機として受注したHITAC 5020システムの重要なソフトウェアであって，社内では開発依頼元の工場から進行状況が予定と違っていることを厳しく責められていたが，我々のマネージャは，そのことを我々には一言もいわなかったので，思う通りに設計を進めることが出来た．そして，無事センタの稼働に間に合わせることが出来た(1965年)．「やる気」があってやっている時には余計なマネージメントをしない方がよいという例であると思う．

3．その後

このプロジェクトで考えたことがその後どうなったかを述べる．

3．1　ループの最適化

図2のプログラムの最内ループ（kのループ）の目的コードを例にとって，ループの最適化がどのように変わってきたかを述べる．HARPの生成するその最内ループの目的コードは8命令である．

```
do i = 1, n
   do j = 1, n
      sum = 0.
      do k = 1, n
         sum = sum + A(i,k)*B(k,j)
      end do
      C(i,j) = sum
   end do
end do
```

図2　行列の掛け算のプログラム

HITAC 5020が完成した頃，IBMは360シリーズを出荷し始めていた．その最初のFORTRANコンパイラ（Gコンパイラと呼ばれていた）の目的コードは13命令であったので，IBMは最初のコンパイラのような最適化はしなくなったのか，あるいは，新しい

360のアーキテクチャでは最適化が難しいのかと思った．しかし，やがて出てきたFORTRAN Hコンパイラのそれは5命令であった（ただし，opt2を指定した場合．opt0では27命令．opt1で20命令）．それは，図3のような命令である（命令は読み安さを考えて変えてある）．HコンパイラはIBMのワトソン研究所で研究開発されたものが製品化されたもので，最適化のためのデータフロー解析などに新しい技術が使われており，その後の最適化コンパイラの技術の基になった．

```
L:  LoadF     R2, A(t1)       ! R2= A(t1)
    MultF     R2, B(t2)       ! R2= A(t1)* B(t2)
    AddF      R1, R2          ! R1(sum)=sum+A(t1)* B(t2)
    Add       t1, t3          ! t1 = t1 + t3
    AddCmBr   t2, 4, t6, L    ! t2 = t2 + 4; if t2 ≤ t6 goto L
```

図3　汎用大型機（CISC）用目的コード

360/370のようなCISCに代わってRISCが登場したときには，命令が単純になって目的コードの命令数が多くなってしまうが，コンパイラの最適化の技術でそれを補うと言われていた．RISCマシンの目的コードとしては，図4のようなものが考えられる．

```
L:  LoadF   R2, A(t1)   ! R2= A(t1)
    LoadF   R3, B(t2)   ! R3= B(t2)
    MultF   R2, R3      ! R2= A(t1)* B(t2)
    AddF    R1, R2      ! R1(sum)=sum+A(t1)* B(t2)
    Add     t1, t3      ! t1 = t1 + t3
    Add     t2, 4       ! t2 = t2 + 4
    Cmp     t2, t6      ! if t2 ≤ t6
    Br      L           ! then goto L
```

図4　RISCマシン用目的コード

しかし，筆者が1990年頃SPARCコンパイラを調べたときには11命令を生成するものしか見つからず，RISCのうたい文句ほど最適化が行われていないのではないかと思った．しかし，1994年に学生に最適化の講義の演習問題として調べてもらったときには，opt2で8命令のものがあった（optなしでは33命令）．さらに，オプションopt3ではループ展開（loop unrolling）も行われ，ループ2回分を展開して14命令にするので，ループ1回あたりでは7命令に相当していた．

RISCマシンはその後，同時に複数個の命令が実行できるスーパスカラマシンへと発展してきている．その場合の最適化では，単に命令数を少なくするだけでなく，命令の順序が問題になる．ループでそのような最適化をするのはソフトウェア・パイプライニングと呼ばれる．その例が図5である．図5でLからL'までがループであり，ロードや掛け算に少し時間がかかっても，図4と比べれば，そのために待たされることがないよう

になっている．（Br命令は遅延ブランチであるとしている）

```
      LoadF   R2, A(t1)      ! R2= A(t1)
      LoadF   R3, B(t2)      ! R3= B(t2)
      Add     t1, t3         ! t1 = t1 + t3
      Add     t2, 4          ! t2 = t2 + 4
L:    MultF   R2, R3         ! R2= A(t1)* B(t2)
      LoadF   R2, A(t1)      ! R2= A(t1)
      LoadF   R3, B(t2)      ! R3= B(t2)
      Add     t2, 4          ! t2 = t2 + 4
      AddF    R1, R2         ! R1(sum)=sum+A(t1)* B(t2)
      Cmp     t2, t6         ! if t2 ≤ t6
      Br      L              ! then goto L
L':   Add     t1, t3         ! t1 = t1 + t3
      MultF   R2, R3         ! R2= A(t1)* B(t2)
      AddF    R1, R2         ! R1(sum)=sum+A(t1)* B(t2)
```

図5　ソフトウェア・パイプライニング

最近のコンパイラがどの程度のソフトウェア・パイプライニングを行っているかは調べていないが，筆者も関係して最近完成した超並列計算機CP-PACS（Computational Physics by Parallel Array Computer Sustem）のFortranコンパイラ[8]では，その要素マシンの目的コードは，4回のループ展開をして14命令となっている．これはループ1回あたり3.5命令に相当する．しかも，スライドウィンドウの機能を利用して，配列が2次キャッシュに入りきらない場合にもロードのために待たされることのないようなコードになっている．なお，命令数が少ないのは，1命令で乗算と加算を行ってしまう命令を利用しているからでもある．

3．2　算術式の最適化

複数のレジスタを最適に利用するアルゴリズムは，プロジェクトが完成してからCACMに投稿した[5]．Sethi等は，このアルゴリズムを基に，その不十分なところを補って完成させた[9]．現在では，このアルゴリズムはSethiとUllmanのアルゴリズムとして知られている．なお，ずっと後になって，Aho[1]は同様のアルゴリズムをErshov[4]が最初に発表していることを発見した．

3．3　論理式の最適化

論理式に対して，飛び越し型（短絡（short circuit）型とも呼ばれる）の最適なコードを1パスで生成するアルゴリズムには誤りがあったが，あまり複雑な論理式を書く人は

居なかったらしく，バグが顕在化することはなかった．正しいアルゴリズムは数年後に完成させた[6][7].

3．4　配列パラメータ

配列の番地計算に掛算の表を使う方式は，コンパイル速度も実行速度も適当に速くするのに役立った．さらに，この方式では，サブルーチンの仮引数として配列が宣言されているときには，実引数の掛算の表（の番地）もパラメータとして引き渡すことにした．配列が引数ならその構造までを引き渡すのは当然と考えたからである．

しかし，IBMの最初のFORTRANをはじめ多くのコンパイラでは実引数の先頭番地だけをパラメータとして引き渡していたらしく，仮引数が配列でも，実引数として配列要素なども書けるようになっていた．FORTRANの米国規格（ANSI）が作られるときにこれが明文化され，それがそのまま国際規格（ISO）にもなったのでJIS FORTRANもそうなってしまった（筆者は反対はしたが）．したがって，この方式はその後のFORTRANには使えなくなった．

その後の言語で，この問題がどう扱われているかを述べておく．

（1）PL/I

PL/Iでは配列の構造の情報をまとめたdope vectorが引き渡される．

（2）Pascal

もとのPascalには配列の長さを引き渡すという考えはなかった．国際規格（ISO）を作るときにそれが出来る整合配列引数が提案されたが，賛否両論があったので，規格にレベルをもうけ，レベル0では整合配列は許されず，レベル1で許されるものとした．

（3）C

Cでは，そもそも配列名はその先頭番地へのポインタを表すものとされる．構造を引き渡すという考えはない．

（4）Ada

配列は，各次元Nについて属性として，先頭のインデックスFIRST(N)，最後の要素のインデックスLAST(N)，長さLENGTH(N)を持っており，それらが受け渡される．

（5）Fortran 90

配列の要素だけでなく，配列そのものを演算の対象とすることもでき，ダイナミックに配列を割り付けることもできる．各次元について，インデックスの下限LBOUND()，上限UBOUND()，長さSIZE()などの情報を得ることが出来る．

（6）Java

番地（ポインタ）を直接扱うという考えはない．配列は，その長さをlengthフィールドとして持っている．インデックスは0から始まる．

4．おわりに

初めにも述べたように，筆者がこの分野に入った時は「自動プログラミングの研究」

という名前でアセンブラの開発が行われていた，ソフトウェアの黎明期ともいうべき時代であったから，何をやっても，それが「新しい仕事」となり得た．その意味で大変幸運であった．その上，上司や同僚にも大変恵まれた環境で仕事をすることができたのも幸せであった．最後に，そのことに感謝したい．

参考文献

[1] Aho, A.V., Sethi, R. and Ullman, J.D.: Compilers -- Principles, Techniques, and Tools, Addison Wesley, 1986.

[2] Backus, J.W. et al.: The FORTRAN automatic coding system, Proc. AFIPS 1957 WJCC, pp.188-198.

[3] Backus, J.W.: The history of FORTRAN I, II and III, SIGPLAN Notices, vol.13, no.8, Proc. History of Programming Languages Conf., pp.165-180, 1978.

[4] Ershov, A.P.: On programming of arithmetic operations, CACM, vol.1, no.8, pp.3-6, 1958.

[5] Nakata, I.: On Compiling Algorithms for Arithmetic Expressions, CACM, vol.10, no.8, pp.492-494, 1967.

[6] 中田育男：論理式のコンパイル方式，情報処理，１６巻，３号，186-194頁，1975.

[7] 中田育男：コンパイラ，産業図書，1981.

[8] 中田育男，山下義行，小柳義夫：超並列計算機CP-PACSのソフトウェア，情報処理，３７巻，１号，29-37頁，1996.

[9] Sethi, R. and Ullman, J.D.: The Generation of Optimal Code for Arithmetic Expressions, JACM, vol.17, no.4, pp.715-728, 1970.