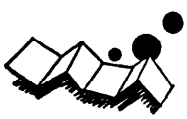
解　説



FORTRAN コンパイラにおける最適化処理†

高　貫　隆　司††

1. はじめに

APL, PASCAL, Ada といった新しいプログラミング言語が次々と登場しているが，科学技術計算の分野では，今なお FORTRAN が主力言語であることに変わりはない．特に，気象，原子力，航空，分子科学のような大規模科学技術計算においては，FORTRAN の占有率は，100% に近いと推定される．これは，ひとつには FORTRAN が昔から使われてきた言語で財産として多くのプログラムが残されていることもあるが，他の言語と比べると FORTRAN コンパイラの最適化レベルが高いことも大きな要因となっている．

FORTRAN は，もともと実用的見地から作られた言語であり，言語仕様そのものも処理系の効率を配慮して決められている．またコンパイラも昔から数多く作られており，高度な最適化機能を備えたものも多い．COBOL や PL/I が主として使われる事務計算では，計算量よりも入出力データ量の方が圧倒的に多く最適化機能が問題になることは比較的少ない．これに比べて，FORTRAN 主体の科学技術計算のプログラムでは，計算量が非常に多く，少しでも速い計算機，少しでも最適化能力の高いコンパイラが要求される．計算能力に対する潜在的要求は非常に強く，現在の最高速コンピュータの 10 倍とか 100 倍といった計算能力を要求している分野もある．これほどではなくても，使用している計算機の能力に合わせて，やむなくモデルを小さくしたり，何回にも分けて計算を進める例は数多くある．このようなユーザにとっては，例えば最適化により 3 割処理時間を短縮できれば，それは 3 割多くの仕事ができることを意味しており，影響は大きい．

本解説では，FORTRAN コンパイラの最適化処理の概要と，それが現実の FORTRAN プログラムにどのような形で適用されて効果を発揮するかということを実例を中心に説明する．さらに一般にスーパコンピュータと呼ばれているベクトルプロセッサ向けの FORTRAN コンパイラの最適化処理についても述べる．

2. 最適化処理の概要

コンパイラの最適化処理技術の基礎はかなり確立しておりそれを紹介した論文や単行本なども数多く出されている[1),2),4)]．以下にこれらの基礎技術についての概要を述べるが詳細は各種文献を参考にされたい．

図-1 に標準的なコンパイラの構成を示す．構文解析では，原始プログラムを入力し，FORTRAN の文法に合った正しい文かどうかを解析する．文法的に正しい文の場合には，中間語と呼ばれるコンパイラ内部で扱い易い表現に変換する．中間語の形式としては四

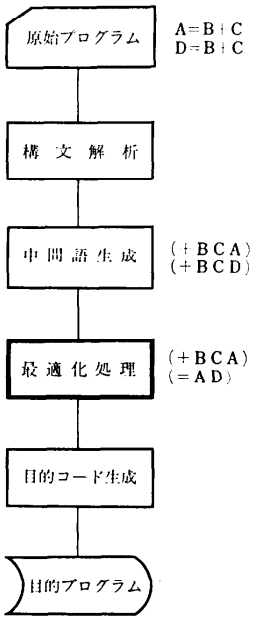


図-1　標準的なコンパイラの構成

† Optimization Techniques of FORTRAN Compilers by Ryuji TAKANUKI (Software Works, Hitachi, Ltd.).

†† (株)日立製作所ソフトウェア工場

つ組 (quadruple), 三つ組 (triple), ポーランド記法 (polish notation) など種々のものがあるが，最適化処理をするコンパイラでは四つ組を使うことが多い[4]．四つ組は演算子，演算対象の2つのオペランド，演算結果のオペランドの4つからなる．たとえば

$A=B*C+D$

を四つ組で表現すると，

$(*BCT)$

$(+TDA)$

という2つの四つ組で表現される．ここでTは演算の中間結果を示す．

最適化処理では，原始プログラムから直接導き出された中間語を入力して，よりよい目的コードが生成できるような再構成された中間語を作り出す．最適化処理は次のような順序で行われることが多い．

- 制御の流れの解析とループの検出
- データの流れの解析
- 最適な中間語の生成
- レジスタ割り付け

(1) 制御の流れの解析とループの検出

一般にプログラムのなかでは，ループ部分の実行比重が大きいので，ループを検出し，その部分を集中的に最適化すれば効果が大きい．FORTRAN の場合には，多くのループは DO ループとして表現されるため，DO ループだけしかループとして検出しないこともあるが，十分な最適化効果を上げるためには，制御の流れの解析によるループ検出処理が必要である．

制御の流れを解析するために，まずプログラムを基本ブロック（制御の移動がない一連の文のかたまり）に分け，基本ブロックの間の制御の流れを有向グラフで表す[1]．**図-2** に有向グラフの例を示す．図-2 の左側の FORTRAN プログラムを有向グラフで示すと右のようになる．図-2 を見るとブロック5からブロック1に戻っているので明らかにループをしている．しかしコンパイラがループを検出する目的は，ループ内の演算をできるだけループの外へ追い出すことにあるのだから，入口が2つ以上あったり，ループの途中へとびこんだりしては困る．例えば，図-2 でブロック3に他の入口があったとすると，このグラフをループとして処理することはできない．なぜならば，例えばブロック4に不変な式があってループ外へ追い出そうとすると，図-2 のグラフでは，ブロック0でその式を実行すればよいが，ブロック3に他の入口がある場合は，ブロック0でその式を実行するわけにはいかないからである．コンパイラでは，ブロックの支配関係（ブロックBに先立ってブロックAが必ず実行される場合，AはBの支配ブロックという）と，ブロックの前後関係をもとにコンパイラが処理可能なループを検出している．DO ループは，入口は必ず1つであり，途中からとび込んだりしないのでつねにコンパイラが処理可能なループを構成する．

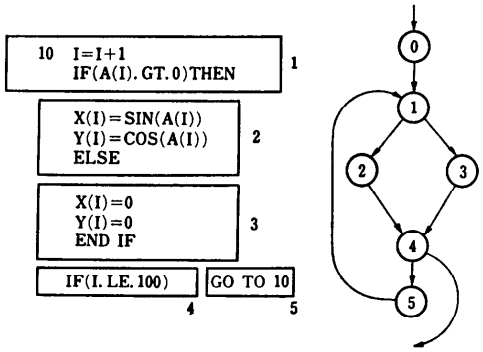


図-2 有向グラフの例

(2) データの流れの解析

データがどこで定義され，どこで参照されるかを解析するのがデータの流れの解析である．データの流れの解析にも種々の手法がある[2]が，有向グラフをもとにデータの定義，参照関係を解析している点では共通している．データの流れの解析結果により，ある場所で定義されたデータがどの路をたどってどこで参照されるか，言いかえればデータは有向グラフ上のどの辺上で生きているか（意味のある値を保持しているか）がわかる．**図-3** は，変数Xの有効範囲はブロック1からブロック3の間およびブロック5以降であり，変数Yはブロック1以降すべてであることを示している．データの定義，参照の位置やデータの有効範囲をもとに，種々の最適化が実施される．

(3) 最適な中間語の生成

原始プログラムを構文解析して作成した中間語のままではあまり効率のよい目的コードは得られない．目的コードの効率を高めるためには，中間語レベルで種種の最適化処理を行い，最適な中間語を生成しなければならない．中間語レベルの最適化は，単純で局所的なものから，複雑で広い範囲を対象とするものまでさまざまであるが，代表的なものとしては次の3つがある．

- 共通部分式の削除

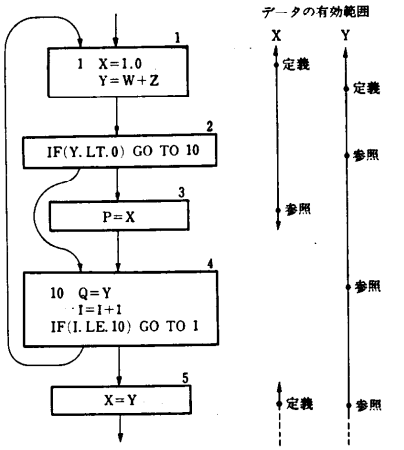


図-3 データの有効範囲

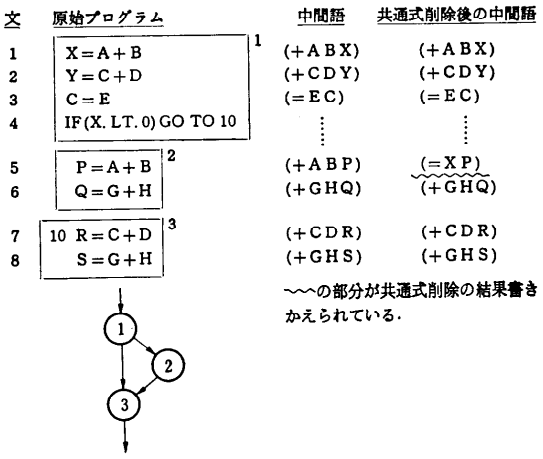


図-4 共通式の削除

- 不変式のループ外への移動
- ループ制御変数の最適化

(a) 共通部分式の削除

同じ結果が得られる式の評価は1回だけで済ませようというのが，共通部分式の削除と呼ばれる最適化である．共通部分式の削除は，中間語上で同じ結果となる演算を探し，2回目以降の演算がなくなるよう中間語を書きかえるという形で行う．**図-4** に共通部分式削除の例を示す．同じ結果が得られる式が共通部分式であるから，共通部分式を見つけるには演算子と演算対象の2つのオペランドがすべて同じものを探せばよい．図-4 の文1と文5の $A+B$ は，同じ結果が得られるので共通部分式である．共通部分式を削除した結果，文5の中間語から加算が削除され単なる代入に置きかえられる．図-4 の $C+D$ と $G+H$ は，演算の形の同じものがあるがどちらも共通部分式ではない．なぜならば，$C+D$ の方は，文3で C の値が変わってしまうため，文2と文7の $C+D$ の演算結果が同じとは限らないためであり，$G+H$ の方は，X の値が負のときは，文6を実行することなく，文8が実行されるので，文6での演算結果をそのまま文8で使うという共通部分式削除の考えがそもそも成り立たないためである．このように制御の流れの解析結果，データの流れの解析結果をもとに同じ値となる式を検出し，削除を行う最適化が共通部分式の削除である．

(b) 不変式のループ外への移動

ループ実行中は結果が変わらない式をループに入る前に演算する最適化を不変式のループ外への移動という．**図-5** に不変式のループ外への移動の例を示す．図-5 の中間語で comp は比較を示す演算子で，BLE は比較後の分岐先を示す演算子である（演算子といっても，四則演算に対応するものだけでなく，プログラム中のすべての操作に対応して演算子がある．またオペランドの数や中間語の形式も演算子によって違う）．また α は分岐先ラベルを示す．不変式の検出は，演算の2つのオペランドがループ中で再定義されないものを探すことにより行う．データがループ中で再定義されるかどうかは，データの流れの解析により解析済であるので不変式の検出は比較的容易である．

(c) ループ制御変数の最適化

ループに対して効率のよい目的プログラムを生成するためには，ループ制御変数に関する処理を最適化することが重要である．ここでループ制御変数とは，ループ中で一定値ずつ変化し，その値によってループの終了を決定するもので，DO 変数は，ループ制御変数の代表的なものである．ループ制御変数の最適化のこと

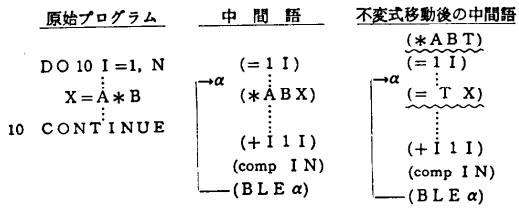


図-5 不変式のループ外への移動

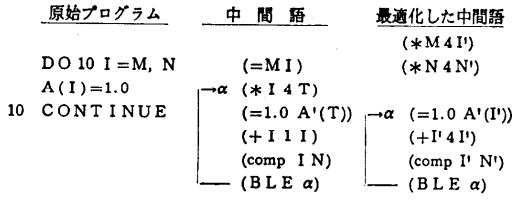


図-6 ループ制御変数の最適化

をストレングスリダクション（演算の強さの軽減）ということもある．これはループ制御変数の乗算を加算に変換することがループ制御変数の最適化の中心であるためである．**図-6** にループ制御変数の最適化を行ったときの中間語を示す．この例のようにループ制御変数は DO ループ内では，配列参照の添字として使われることが多い．1つの配列要素の大きさを4バイトとすると，配列参照 $A(I)$ の位置決めのためのアドレス計算は，$A(1)$ が A の先頭であるから

(Aの先頭アドレス)+($I-1$)*4

となる．(Aの先頭アドレス)−4 はコンパイル時点で決定できるのでそれを A' とし，$I*4$ の結果を T とすると $A(I)$ のアドレスは $A'+T$ となる．図-6 ではこれを $A'(T)$ と表現している．ストレングスリダクションは，このような配列参照のアドレス計算に現れる乗算を加算に変換する最適化である．図-6 で，I が1ずつふえると $I*4$ は4ずつふえることを利用し，I の4倍の値を持つ変数 I' を導入する．I' を使うとループ内の $I*4$ は I' と置きかえることができる．ループの最後では，$I'=I'+4$ という中間語を新たに挿入し，I と N との比較を I' と N' ($N*4$ の値を持つ新たな変数）との比較におきかえる．ループ内で I は，$I*4$ の形でしか使われていないので，$I=I+1$ という中間語は削除する．こうして最適化した中間語をもとの中間語と比較すると，ループ内の乗算が1つ減ってかわりにループの外で新たに2つ乗算が増えていることがわかる．M, N が定数の場合には，乗算をコンパイル時に実行できるので，ループ外の乗算の増加はなくなる．

（4） レジスタ割り付け

ハードウェアのアーキテクチャによっても異なるが一般に計算機は，レジスタと呼ばれる比較的少数の高速メモリを持っており，これの使い方が目的プログラムの効率に大きく影響する．以下では HITAC M シリーズのアーキテクチャをもとに説明を行う．HI-TAC M シリーズは16個の汎用レジスタと4個の浮動小数点レジスタを持つ．汎用レジスタは，ベースレジスタ，インデックスレジスタおよび固定小数点演算レジスタとして使用する．浮動小数点レジスタは，浮動小数点演算用のレジスタとして使用する．以下の記述で GR は汎用レジスタを，FR は浮動小数点レジスタを示す．

レジスタ割り付けには一般的アルゴリズムはなく，種々の要因を加味して最適と思われる割り付けが行われる．レジスタ割り付け処理は大別すると次の3つに分かれる．

- ループに対する大域的なレジスタ割り付け
- 局所的なレジスタ割り付け
- 大域レジスタの再割り付け処理[3]

（a） 大域的レジスタ割り付け

レジスタ割り付け処理もループ単位に行われる．大域的レジスタ割り付けとは，ループ全体を通してあるレジスタを特定目的用として使用することを意味する．大域的レジスタ割り付けの対象としては，ベースアドレス用レジスタ，ループ制御用レジスタ，変数，不変式などがある．大域的レジスタ割り付けをしてしまうとそのループでは他の目的に使えなくなってしまうので，使用頻度等の重みをつけて，最も効果的なものに割り付けられるよう工夫がなされる．

（b） 局所的レジスタ割り付け

ブロック内の中間語に対して最も効率のよい目的プログラムが生成されるようなレジスタ割り付けを行う．効率のよいレジスタ割り付けとは，メモリ参照ができるだけ少なくてすむような割り付けということである．たとえば値を代入した変数が同じブロックの後方で参照されるときは，参照時点までレジスタに値を残すような割り付けを行う．

（c） 大域レジスタの再割り付け処理

大域レジスタを割り付けるとループ内では別の目的には使えなくなる．しかしながら変数に対して割り当てたレジスタは，ループ全体に括りつける必要がない場合には，部分的に別目的に使用することが可能である．たとえば，**図-7** で変数 S に対して大域的に割り付けたレジスタは，①と③の区間では変数 S の値を保持する必要があるが，②の部分では，S の値を保持し

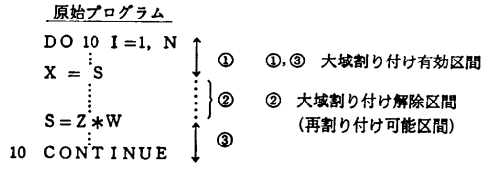


図-7 大域レジスタの再割り付け

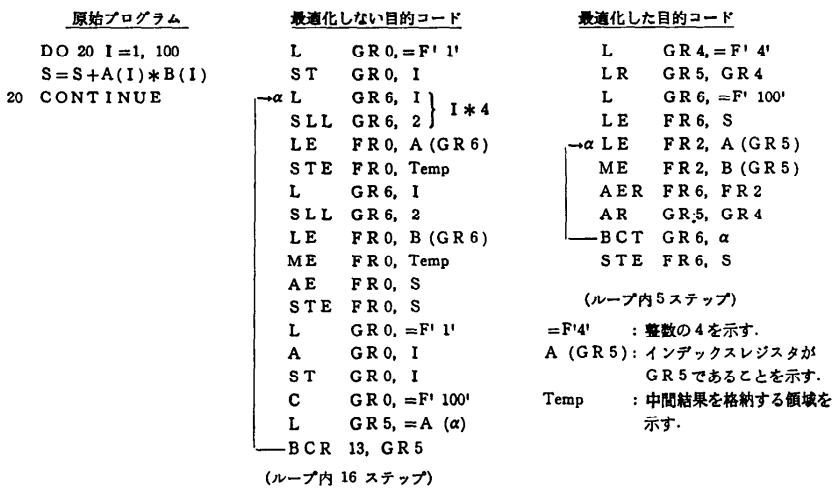


図-8 最適化目的コードの例

なくてもよいので別の目的に再割り付けしてもよい.

以上, (1)から(4)に述べた最適化処理の結果生成される目的コードの例を**図-8** に示す. 最適化しないときは, 添字計算として $I*4$ (命令では L 命令と SLL 命令) の計算が2回あるが, 右の最適化した目的コードでは共通部分式の削除で1回に減り, さらにストレングスリダクションにより, その1回の乗算もなくなっている. また大域的レジスタ割り付けにより, ループ内の命令数が大幅に少なくなっている. GR5 は $I*4$ の値をもつ I' に, GR4 は整定数4に, FR6は変数 S にそれぞれ割り当てられている. また, GR6にはループ回数が割り当てられており, BCT 命令1つでループを制御している (BCT 命令は, レジスタの内容を1ずつ減じて0になるまではオペランドで示したアドレスへ分岐する命令である). このように種々の最適化が組み合わさることにより, 最適化する前の 16 ステップが5ステップと3分の1以下に削減される.

3. 最適化処理の違いによる目的コードの差

同じような最適化をしていても, 処理方式が異なれば目的コードに差が出てくる. **図-9** は, DO ループに対する目的コードが最適化によりどのように違ってくるかを示している. 目的コード例1では, $K*4$ がストレングスリダクションされており, $B(K, I)$ および $B(K, J)$ の添字計算は, 配列参照の直前で実行される. これに対し, 目的コード例2では, $B(K, I)$ の添字 $(K*4+I*400)$ と $B(K, J)$ の添字 $(K*4+J*400)$ がそれぞれ別々にストレングスリダクションされており, 配列参照の直前で添字計算は出ない ($K*4+I*400$ の式は, ループ1回ごとに一定値ずつふえるのでストレングスリダクションが可能である). 目的コード例3では, ストレングスリダクションの対象は $K*4$ であり, 例1と同じであるが, ループ内のステップは例1より4ステップ少ない. これは, 例1では添字計算をループ内で行っているのに対し, 例3では, 添字の中でループ内では不変な部分 ($I*400$ と $J*400$) をあらかじめ配列 B のアドレスから引いておき, それをベースアドレスとして大域的レジスタ割り付けを行うことによりループ内での添字計算のステップを削減しているためである[3]. 図-9 に示すようにいずれもストレングスリダクションや大域的レジスタ割り付けなどの最適化を行っているが, 処理が異なれば, 生成される目的コードの効率は大きく異なってくる. 特に例1では, レジスタコンフリクトと呼ばれる現象のために, みかけのステップ数の差以上に目的コードの効率は悪くなっている. **図-10** は, レジスタコンフリクトを示したものである. パイプライン制御を行う計算機では, 命令を D, A, OF, E の4つのステージに分け, Eサイクル(実行サイクル)が次々と連続するように制御することによって高速化を実現している. ところが, 図-10 の LE 命令のように, オペランドアドレス計算に必要なレジスタの値が直前の命令で決まる場合は, 次の命令の Dステージは, 直前の命令の実行完了まで待たされる. これをレ

原始プログラム

```
      REAL  B (100, 100)
      DO  10  K=1, 100
      SUM=SUM+B (K, I)*B(K, J)
   10 CONTINUE
```

目的コード例1

```
→α LR   GR1, GR3
  AR   GR1, GR4
  LE   FR2, B(GR1)
  LR   GR1, GR3
  AR   GR1, GR5
  ME   FR2, B(GR1)
  AER  FR6, FR2
  AR   GR3, GR2
  BCT  GR6, α
```

レジスタ割り当て
GR2: 4
GR3: K*4
GR4: I*400
GR5: J*400
GR6: ループ回数
FR6: SUM

目的コード例2

```
→α LE   FR2, B(GR3)
  ME   FR2, B(GR4)
  AER  FR6, FR2
  AR   GR3, GR2
  AR   GR4, GR2
  BCT  GR5, α
```

レジスタ割り当て
GR2: 4
GR3: K*4+I*400
GR4: K*4+J*400
GR5: ループ回数
FR6: SUM

目的コード例3

```
→α LE   FR2, B-I*400(GR3)
  ME   FR2, B-J*400(GR3)
  AER  FR6, FR2
  AR   GR3, GR2
  BCT  GR4, α
```

レジスタ割り当て
GR2: 4
GR3: K*4
GR4: ループ回数
FR6: SUM
この他に B-J*400 および B-J*400 のベースアドレスレジスタが割り当てられている.

図-9 最適化処理の違いによる目的コードの差

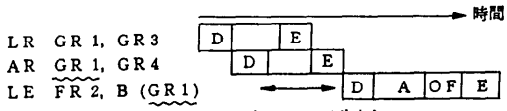


図-10 レジスタコンフリクト

ジスタコンフリクトといい，Eステージが連続する場合と比べるとコンフリクトを起こした命令の処理時間は数倍に伸びる．このレジスタコンフリクトを考慮に入れると図-9 の例3の目的コードは例1より 2～3 倍も効率がよいことになる．

図-9 の目的コード例3は，かなり最適に近いコードのように見えるが，これよりもさらに効率のよい目的コードも可能である．図-11 の例4は，図-9 のプログラムのループ回数を半分にすることによりさらに効率化を計ったものであり，次の原始プログラムと等価なオブジェクトを生成する最適化である．

```
      DO 10 K=1, 100, 2
      SUM=SUM+B(K, I)*B(K, J)
      SUM=SUM+B(K+1, I)*B(K+1, J)
   10 CONTINUE
```

図-11 の例4のループ1回（8ステップ）は，例3のループ2回分（10 ステップ）に対応するので，ループ2回で2ステップ分，例4の方が実行ステップ数が少ない．

原始プログラム

```
      REAL  B (100, 100)
      DO  10  K=1, 100
      SUM=SUM+B (K, I)*B (K, J)
   10 CONTINUE
```

目的コード例3

```
→α LE   FR2, B-I*400 (GR3)
  ME   FR2, B-J*400 (GR3)
  AER  FR6, FR2
  AR   GR3, GR2
  BCT  GR4, α
```

GR4: ループ回数 100

目的コード例4

```
→α LE   FR2, B-I*400 (GR3)
  ME   FR2, B-J*400 (GR3)
  AER  FR6, FR2
  LE   FR2, B-I*400+4(GR3)
  ME   FR2, B-J*400+4(GR3)
  AER  FR6, FR2
  AR   GR3, GR2
  BCT  GR4, α
```

GR4: ループ回数 50

図-11 ループ回数半減による最適化

4. 種々の最適化

以上述べた他にもさまざまな最適化が行われる．そのいくつかを次に説明する．

4.1 ハードウェアに依存する最適化

(1) 命令の並びかえ

前述したように，パイプライン制御を行うハードウェアでは，レジスタコンフリクトをさけることで実行性能がかなり向上する．図-12 に示すようにベースレジスタまたはインデックスレジスタを更新する命令と，それをベー

```
LE  FR 2, 0(GR 4, GR 7)      ¥  L   GR14, 552 (0, GR13)
SE  FR 2, 108 (0, GR13)     ⇒   LE  FR 2, 0(GR 4,GR 7)
L   GR14, 552 (0, GR13)         SE  FR 2, 108 (0, GR13)
ME  FR2, 0 (GR14, GR 6)         ME  FR 2, 0(GR14,GR 6)
```

¥：移動した命令を示す

図-12 命令の並びかえ

スまたはインデックスとして使用する命令との間を離す最適化である．

（2） 除算の乗算への転換

最近の高速な計算機では，乗算は非常に速くなってきており，他の命令と大差なくなってきているが，除算命令は相変わらず遅い．このため，ループ内に除数が一定の浮動小数点除算があれば乗算にかえる最適化である（精度が変わってくるので無条件にはできない）．

```
      DO 10 I=1, N              X=1.0/C
10    A(I)=B(I)/C      ⇒        DO 10 I=1, N
                        10      A(I)=B(I)*X
```

4.2 実行時サブルーチンに関する最適化

FORTRAN コンパイラの目的プログラムを実行するときには，コンパイラが生成する目的コード部分だけでなく，組込み関数や入出力処理などの実行時サブルーチンの実行比率もかなり大きいことが多い．組込み関数や入出力処理の高速化も重要な最適化である．

（1） 組込み関数のインライン展開

組込み関数の高速化のためには，関数のアルゴリズムを改良することが第一であるが，アルゴリズムに関係のない目的コードとのインタフェースがかなりのオーバヘッドになっていることも事実である．このインタフェースオーバヘッドを減らすために，組込み関数を引用場所に直接展開する最適化である．

（2） DO 形並び入出力文の最適化

DO 形並びがあると，目的コード側でループし，ループの1回ごとに実行時サブルーチンが呼ばれる．DO 形並び入出力文の最適化は，DO 形の構造（初期値，終値，増分値，ネスト数など）を実行時サブルーチンに渡し，実行時サブルーチン内でループさせることにより，目的コードとのインタフェースオーバヘッドを削減するものである．

4.3 プログラム構造の変更を伴う最適化

もとのプログラムのループ構造などを変えてしまうような最適化もある．コンパイラの最適化として実現しているものもあれば，ソースオプティマイザとしてプリプロセッサの形で実現しているものもある．

（1） 多重 DO ループの一重化

2重以上のループを1重のループとみなして処理する最適化である．完全入れ子の DO ループで配列はすべての要素を参照していなければならないなどきつい条件がつくことが多い．

（2） DO ループ回数の半減

DO ループ内の文をループ2回分に展開して書き，ループ回数を半分にする最適化である．

```
      DO 10 I=1, 100              DO 10 I=1, 100, 2
10    A(I)=B(I)          ⇒        A(I)=B(I)
                         10       A(I+1)=B(I+1)
```

初期値，終値の値によっては，最終回ループだけ特別に処理する必要がある．また，ループ回数が少ない場合や，ループ内の文が多すぎる場合には，最適化によって逆に遅くなることもある．

（3） DO ループの展開

ループ回数が小さい DO ループは，ループにしないで展開してしまう方がよい．

```
      DO 10 I=1, 3              A(1)=B(1)
10    A(I)=B(I)          ⇒      A(2)=B(2)
                                A(3)=B(3)
```

特にこのようなループが2重ループの内側にある場合は，展開により外側ループが最内側ループに変わって最適化の対象となるので効果は大きい．

（4） 手続きの引用場所への展開

小さなサブルーチンや関数を引用している場所に直接展開して，手続き引用のオーバヘッドを削減する最適化である．複数のプログラム単位を処理の対象としなければならないのでコンパイラでこの最適化をやっている例はなく，プリプロセッサとして実現しているのが普通である．つねに効果がある最適化とは言えないがプログラムをチューニングする過程でこのような最適化を選択的に実行したくなることはよくある．

4.4 その他の局所的最適化

その他の局所的最適化としては次のようなものがある．

（1） 定数計算，定数型変換のコンパイル時計算
（2） 整定数べき乗の乗算での実行
（3） 文関数の引用場所での展開
（4） 論理 IF 文の論理式の最適化

演理演算子がすべて .AND. であるとき，1つでも偽のものがあれば全体を偽として評価する．また，すべて .OR. のときは，1つでも真のものがあれば全体を真として評価する．

（5） 算術 IF 文における不要な分岐命令の削除

5. ベクトル演算と最適化

ベクトル演算命令を持つアレイプロセッサ向けの FORTRAN コンパイラの最適化は，汎用計算機の場合とはかなり異なってくる．汎用計算機の場合は，ループ内の演算をいかに少ない命令(基本オペレーションの集まり)で実行するかが最適化の中心課題となる．これに対してアレイプロセッサでは，どれだけ多くのループをベクトル命令で実行できるかが問題となる．なぜならばベクトル命令は，それと同等の機能を通常の命令（以後スカラ命令とよぶ）で実行する場合と比べると数倍から数十倍速く，ベクトル命令で実行できる割合（これをベクトル化率という）が性能の決定要因となるからである．ベクトル命令は，ベクトルどうしの演算に対して用意された命令であり，FORTRAN プログラム中のベクトル演算部分はベクトル命令で実行し，それ以外の部分はスカラ命令で実行する．FORTRAN 言語には，ベクトル演算を直接表現する機能はなく，通常は DO ループでベクトル演算を表している．しかし，DO ループは任意のループを作ることができるので，DO ループがすべてベクトル演算をしているわけではない．FORTRAN コンパイラがプログラム中の DO ループを解析してベクトル演算命令を使った目的コードを作りだすことを自動ベクトル化という．**図-13** は，HITAC M-280 HIAP（内蔵アレイプロセッサ）用の最適化 FORTRAN 77 コンパイラが出力したベクトル命令目的コードの例である．ベクトル A とベクトル B の内積を計算しているループであるということをコンパイラが認識して，ベクトル内積命令である VIPE 命令を用いた目的コードを生成している．

DO ループがベクトル化できるかどうかという認識は，概略次のような手順で行う．

（1） ループ内の文の種類による判定

原始プログラム

```
      DO  10  I=1, 100
10    S=S+A(I)*B(I)
```

スカラ命令による目的コード		ベクトル命令による目的コード	
LE	FR4, S	LE	FR0, S
→α LE	FR2, A(GR8)	SR	GR0, GR0
ME	FR2, B(GR8)	L	GR1, =F'100'
AER	FR4, FR2	LA	GR15, OAV 001
AR	GR8, GR10	VIPE	0, 15
BCT	GR11, α	↑ ベクトル命令	
STE	FR4, S	STE	FR0, S

図-13 ベクトル命令目的コードの例

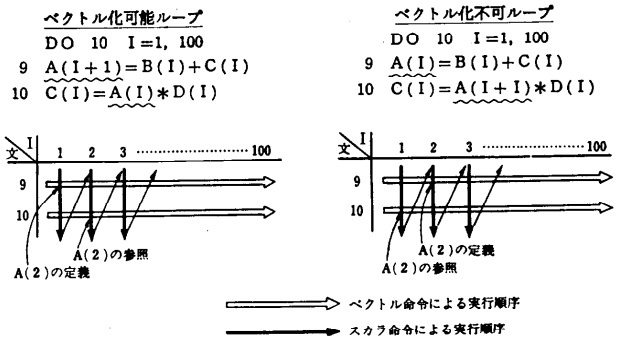


図-14 データの定義，参照関係によるベクトル可否

（2） 型や演算の種類による判定

（3） データの定義と参照の関係による判定

(1)と(2)については，IF 文を含むループのようにアレイプロセッサが持っているベクトル命令の種類に依存して判定が左右される場合と，ループ外へのとび出しや入出力文を含む DO ループなどのように，もともとベクトル演算とは関係がなくどのアレイプロセッサでも不可と判定される場合とがある．(3)のデータの定義参照関係を示したものが**図-14** である．スカラ命令の場合は，I の値を1ずつ上げていきながら文9と文 10 を交互に実行していくのに対し，ベクトル命令では，文9を I の1から 100 まで一度にまとめて実行し，次に文 10 をまとめて実行するというように実行順序が変わってくる．このように実行順序が変わって結果に変わりがなければ，ベクトル化可能であり，結果が異なるようであればベクトル化不可と判定する．結果が同一かどうかは，同一データ領域に対しての定義，参照の前後関係で判定できる．ベクトル命令による実行順序の変更に対して，定義，参照の前後関係に変わりがなければベクトル化可能と判定する．図-14 で配列Aの定義参照関係をみると，左側は，スカラ命令でもベクトル命令でも定義が先で参照が後という関係にあるためベクトル化可能である．一方，右側の DO ループでは，スカラ命令のときは参照，定義の順であるが，ベクトル命令では定義が先で参照が後と逆の関係になるためベクトル化できない（A(2)の定義，参照位置関係に着目）．

6. ま と め

HITAC S-810 や FACOM VP-200 などのスーパコンピュータが発表されたり，HITAC M-180/M-200H/M-280H IAP（内蔵型アレイプロセッサ）や ACOS-1000 IAP（統合アレイプロセッサ）など

のように汎用機にベクトル演算機構を組み込んだものが一般に使われるようになってきたことから，高速計算を要求する分野では，これまでの汎用コンピュータ中心からアレイプロセッサ中心へと変わりつつある．FORTRAN コンパイラの最適化も従来のスカラの最適化に加えて，アレイプロセッサの高速機能を引き出すベクトル最適化が重要になってきている．しかしながら，アレイプロセッサといってもすべてベクトル命令で実行できるのではなく，スカラ命令の実行もかなりの割合で残る．スーパコンピュータのようにベクトル命令が速くなればなるほど，残ったスカラ命令の性能に及ぼす影響は大きくなる．スカラ命令レベルでのループの最適化技術はかなり高度なレベルに達しているが，ベクトル化率を向上させる技術，ベクトル化されない部分のスカラ処理の効率化などは，FORTRAN コンパイラの最適化にとっての今後の課題と考えられる．

参 考 文 献

1) Aho, A. V. and Ullman, J. D.: *Principles of Compiler Design*, Addison Wesley (1977).
2) Allen, F. E. and Cocke, J.: A Program Data Flow Analysis Procedure, CACM, Vol. 19, pp. 137–147 (1976).
3) Scarborough, R. G. and Kolsky, H. G.: Improved optimization of FORTRAN object programs, IBM J. Res. Develop., Vol. 24, No. 6, pp. 660–676 (1980).
4) 中田育男：コンパイラ，産業図書 (1981).
5) Takanuki, R., Nakata, I. and Umetani, Y.: Some Compiling Algorithms for an Array Processor, Proc. 3rd USA-JAPAN Comp. Conf. pp. 273–279 (1978).

(昭和 57 年 12 月 16 日受付)