Vol. 37 No. 6 情報処理学会論文誌 June 1996

多重ループにおける最適ループ展開数算定技法

古関 聰[†] 小松秀昭[††] 深澤良彰[†]

プログラムの実行時間においてループ部分の占める割合は高く，ループ展開による最適化の効果は高い．従来は，最内ループのみの展開を行っており，多重ループが持つ高い並列性を引き出したり，多重ループ間における値の再利用を十分に行うことができなかった．入れ子になっている複数のループを同時に展開することにより，並列度の抽出が可能な範囲と値を再利用できる範囲を飛躍的に大きくすることができる．したがって，最内ループのみではなく，より外側のループも同時に展開したほうが，より高い並列化の効果が期待できる．並列化の効果は，対象となるループの繰返し間の依存関係および値の再利用の関係により決定され，最内ループのループ本体をどのようなループの組合せで展開するか（ループの展開方向），およびループを何回展開するか（ループの展開数）によって得られる効果が異なる．また，ターゲットマシンのリソースは有限であるため，展開の効果は飽和し，それ以上展開しても効果が得られない展開数が存在する．したがって，最大の効果を得るためには，繰返し間の依存および値の再利用，マシンリソースの各情報を十分に考慮し，最適なループの展開方向と展開数を決定しなければならない．本稿では，プログラムから依存や再利用の情報を獲得および表現する方法を確立し，それらとマシンリソースの情報を利用して，多重ループにおける最適に近いループ展開数と方向を見積る方法を述べる．

A Method for Estimating Optimal Times of Unrolling for Nested Loops

Akira Koseki,[†] Hideaki Komatsu[††] and Yoshiaki Fukazawa[†]

Loop unrolling is one of the most promising parallelization techniques, because programs have characteristics that most processing time is spent in their loops. Existing methods would unroll only the innermost loop, so they cannot derive high parallelism, which nested loops originally have, and cannot fully utilize the reuse of data. With unrolling nested loops, the scope, in which data can be reused and instructions can be parallelized, is largely expanded. Therefore, we will obtain much parallelism if we unroll not only the innermost loop but also outer loops. The efficiency obtained by unrolling nested loops is affected by the dependencies among iterations of a loop and the reuse of data, therefore, the direction of unrolling (which loops should be unrolled) and the times of unrolling (how many times a loop should be unrolled) extremely affect the efficiency of the unrolled program. Additionally, the effect of unrolling will saturate because the resources of a hardware are limited, therefore, the certain times of unrolling exists where no more effect can be obtained by unrolling the loops. Consequently, to obtain the best effect, we have to decide the times and the directions of loops unrolling based on the information such as the dependencies among iterations, the reuse of data and the machine resources. In this paper, we described a method to obtain and express such information as dependencies and reuse, and a heuristic algorithm to decide the times and the directions of loops unrolling by utilizing this information and machine resource's one.

1. はじめに

プログラムのループ部分の実行時間は，全体の実行時間に占める割合が大きいので，この部分の最適化は実行時間全体の短縮に大きな効果がある．特に，数値演算ループのほとんどは，繰返し間の依存関係による並列実行の制約が小さいので，ベクトル型計算機や命令レベル並列計算機による演算の並列実行が容易であり，今までに多くの最適化研究がなされている[1)~3)]．

ループ最適化の1つに，プログラムのループ部分において，各繰返しを展開する手法がある．この最適化を行うと，ループ制御に相当する条件判定とジャンプの命令が各繰返しごとに省略されるだけでなく，ジャンプによる命令パイプライン分断が抑制されるので，

† 早稲田大学理工学部
School of Science & Engineering, Waseda University
†† 日本IBM（株）東京基礎研究所
Tokyo Research Laboratory IBM Japan, Ltd.

得られる効果が大きい．特に，繰返し間の制御依存が解消され，繰返しを越えた命令の移動が可能になることは，最も重要である．すなわち，スーパスカラ，VLIW 等の命令レベル並列計算機にとっては，命令の移動により並列実行の機会が広がり，その結果，プログラムの並列度を大きく引き出すことが可能になる．また，連続した繰返しにおいて，配列の同じ要素を参照したり，同じ値を計算する場合が多く，繰返しを展開して1つにまとめると，それらの値生成は共通部分式となる．したがって，展開ループ本体において，その値生成に対応するメモリの参照や演算が一度で済み，残りはその値を格納するレジスタへの参照に置き換えられる．その結果，ループ全体の命令実行回数を削減でき，実行時間を短縮できる．

ループ展開にはこのような効果があるため，命令レベル並列計算機を対象にして，ハードウェアの制約とソフトウェアの制約の比較により最適なループ展開回数を決定する手法がいくつか提案されている[4),5)]．しかし，これらの手法では，ループ展開の対象は最内ループに限られている．一方，全可換（fully permutable）な多重ループ[6)]においては，どのループを選んで展開することも可能であるだけでなく，複数のループを同時に展開することで，繰返し間依存のウェーブフロント[6)]の進行方向における並列度の抽出が可能になったり，値を再利用できる範囲をさらに広げることができる．このため，最内ループだけでなく，より外側のループも同時に展開したほうが，より高い実行高速化の効果が期待できる．

多重ループの展開を考えた場合，得られる並列化の効果は，対象となるループの繰返し間の依存関係および値の再利用の関係により決定され，最内ループのループ本体をどのようなループの組合せで展開するか（ループの展開方向），およびループを何回展開するか（ループの展開数）によって得られる効果が異なる．したがって，最適なループ展開回数と方向を決定するには，繰返し間にまたがる依存および値の再利用の関係を知る必要がある．このような情報を得るためには，コンパイル時にループ構造の命令間依存解析を行い，配列参照の添字を比較して，ループの繰返し間における依存関係および再利用関係を計算することが必要となる．

また，ターゲットマシンのハードウェアリソースは有限であるため，展開の効果は飽和し，それ以上展開しても効果が得られない展開数が存在する．したがって，効率的な展開を行うためには，マシンリソースの情報を利用し，ループをある方向に展開した場合にどの回数で効果が飽和するかを把握しなければならない．

本稿では，以下の順で，我々が提案する多重ループ展開の技法を説明していく．まず，多重ループの多次元展開にはどのような効果があるか，またどのような制約があるかを示す．次に，多重ループの展開回数を決定するのに必要な情報，特に依存情報および再利用情報を抽出する方法を説明する．最後に，これらの情報ならびにマシンリソースの情報を用いて最適な各ループの展開回数と方向を見積るアルゴリズムを示し，これによるプログラムの実行性能の向上を評価する．

2. 多次元展開の性質

2.1 多次元展開の実際

多次元展開の例を以下に示す．まず，**図1** のループ本体に対するプログラム依存グラフ[7)]を**図2** に示す．このプログラムは，2重のループから構成されており，ループ本体は添え字 **i** が増える方向（**i** 方向），および添え字 **j** が増える方向（**j** 方向）に同時に展開できる．

次に，図1のループを **i**, **j** 方向とも展開するために，たとえば **i** について2回，**j** について3回の繰返しをひとまとめにして，もとのループから分離する．これを**図3** に示す．

この段階では，単に1個のループを2個に分割しただけで，各繰返しの実行順序は変わっていない．このループにおいて，**i** のループと **jj** のループを交換し，6回の繰返しからなる2重のループを得る（**図4**）．本

```
for i := 1 to IMAX do
  for j := 1 to JMAX do
    D[i] := D[i] + A[i,j] * (B[j] + C[j]);
```

図1 対象プログラム
Fig. 1 Target program.

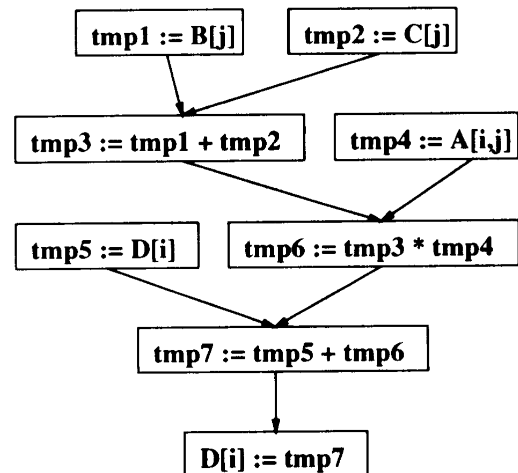

図2 ループ本体の依存グラフ
Fig. 2 Dependence graph of loop body.

```
for ii := 1 to IMAX step 2 do
  for i := ii to ii + 1 do
    for jj := 1 to JMAX step 3  do
      for j := jj to jj + 2 do
        D[i] := D[i] + A[i,j] * (B[j] + C[j]);
```

図3 ループインデックスの分割
Fig. 3 Division of loop indices.

```
for ii := 1 to IMAX step 2 do
  for jj := 1 to JMAX step 3 do
    for i := ii to ii + 1 do
      for j := jj to jj + 2 do
        D[i] := D[i] + A[i,j] * (B[j] + C[j]);
```

図4 ループのブロック化
Fig. 4 Loop blocking.

```
for i := 1 to IMAX step 2 do
  for j := 1 to JMAX step 3 do
  begin
    D[i] := D[i] + A[i,j] * (B[j] + C[j]);
    D[i] := D[i] + A[i,j+1] * (B[j+1] + C[j+1]);
    D[i] := D[i] + A[i,j+2] * (B[j+2] + C[j+2]);
    D[i+1] := D[i+1] + A[i+1,j] * (B[j] + C[j]);
    D[i+1] := D[i+1] + A[i+1,j+1] * (B[j+1] + C[j+1]);
    D[i+1] := D[i+1] + A[i+1,j+2] * (B[j+2] + C[j+2]);
  end
```

図5 多次元展開
Fig. 5 Unrolling nested loops.

論文では，この部分をブロック，また，この処理をブロック化と呼ぶ．この操作は，もとの **i** と **j** のループが互いに交換可能であること，つまり，この入れ子間の交換を妨げるような繰返し間の依存関係が存在しないとき，かつそのときに限り実行可能である．

最後に，ブロックをなす入れ子ループ内のすべての繰返しを展開し，1つのループ本体にする（**図5**）．この，図1から図5までのプログラム変形が多次元展開である．この例では，**i** について2回，**j** について3回のループ本体を展開し，もとのループの6回分の繰返しからなる新しいループ本体が生成されている．

2.2 展開後のループ本体の最適化

次のステップとして，展開されたループ本体中の並列性の抽出と生成された値の再利用を行い，ループ本体の最適化を進める．

まず，配列要素 **B[j]**，**B[j+1]**，**B[j+2]** を参照する命令はそれぞれ2回存在し，しかも各々はつねに同じ値を生成することが分かっているので，1回目の配列要素の参照に限りロード命令を用い，その値を適当なレジスタに確保し，2回目以降は，このレジスタの値を使えばよい．このようにして，以上3個の配列要素に対するロード命令をそれぞれ省略でき，展開後のループ本体から命令を3個減らすことができる．同様に，配列 **C** についても3個のロード命令を減らすことができる．

生成された値の再利用による命令削減は，ロード命令だけにとどまらず，四則などの演算にも現れることがある．この例では，値 **B[j]+C[j]** の生成は，2回出現するが，いずれも同じ値になることが分かっている．そこで，1回目の値生成のときに実際に加算命令を実行し，その値をレジスタに残し，2回目では，そのレジスタの値を用いることで1個の加算命令を省略する．同じことが **B[j+1]+C[j+1]**，**B[j+2]+C[j+2]** についても成り立ち，合計3個の加算命令を減らすことができる．

配列 **D** については，配列の参照命令間に依存関係が存在するのでやや複雑になる．すなわち，1回目の左辺の **D[i]** は，2回目の右辺の **D[i]** に対してフロー依存をなしているが，2回目の右辺の **D[i]** の値は1回目の左辺の **D[i]** の値とつねに等しいので，1回目のストア命令のソースレジスタで置き換え，2回目のロード命令を省略する．このような置き換えによる最適化は，一般にスカラリプレースメントと呼ばれる．同様にして，3回目のロード命令も省略できる．

また，1回目の左辺の **D[i]** と2回目の左辺の **D[i]** の間には出力依存が存在し，しかも1回目の左辺の **D[i]** の値はつねに2回目の左辺の **D[i]** により上書きされる．したがって，2回目の右辺の **D[i]** をレジスタに置き換えた後では，1回目のストア命令でメモリに書かれた **D[i]** の値は，一度も使わないので，このストア命令は無駄になり省略できる．

また，配列 **D** の参照以外では，命令間の依存関係は存在しない．したがって，互いに依存関係のない **A[i,j]**，**A[i,j+1]**，**A[i,j+2]**，**A[i+1,j]**，**B[i+1,j+1]**，**B[i+1,j+2]**，**B[j]**，**B[j+1]**，**B[j+2]**，**C[j]**，**C[j+1]**，**C[j+2]** に対するロード命令などを，順番を並べ替えて並列に実行することができる．

以上が多次元展開後のループ本体の最適化である．これは，**i** 方向に2回，**j** 方向に3回展開を行った例であるが，命令の並列化や値の再利用は各方向に任意の回数で展開を行ったときも，同様に論ずることができる．また，以上の分析から，どの命令がどのように並列化されるか，およびどの値がどのように再利用されるかは，展開方向と展開回数をパラメタとして形式化が可能である．たとえば，配列 **B** に関しては，**j** 方向に展開されたそれぞれのロード命令は **i** 方向の展開回数だけその値が再利用され，**j** 方向の展開回数だけ並列化される．効率のよい多次元展開を行うためには，

このような再利用や並列化の行われ方を十分に把握する必要がある.

3. プログラムの性質の抽出

本章では，前章で述べた多次元展開後のループ本体の最適化をどのように表し，どのような情報を利用するかについて述べる.

3.1 依存および再利用情報の検出

ループ本体に対し大域的データ流れ解析を行い，プログラム依存グラフ[7]を作成する．また，再利用ベクトル[8),9)]と，繰返し間依存[4]を解析して，以下のような具体的な情報の抽出を行う.

3.1.1 繰返し間依存情報の検出

ループの実行時においては，ある繰返しで生成された値と，それ以降の繰返しにおいて生成された値が依存関係を持つ場合がある．この依存関係を繰返し間依存と呼ぶ.

繰返し間依存の検出にあたっては，依存を生じる可能性のあるものはすべて網羅しないと，実行時の結果が異なってしまうおそれがある．しかし，実行時において依存関係の発生しないものも取りあげてしまうと，最適化を妨げてしまうことになるので，できるだけ正確なデータ流れ解析をする必要がある．正確な流れ解析を行うためには，配列参照に関する正確な情報が必要である．そこで，本論文では，配列参照の情報として，

(1) 配列名
(2) 各添字の値を与える多項式

を把握する.

繰返し間依存の表記については，繰返し間のフロー依存，逆依存，出力依存を

- **flow dependence**$(\mathbf{n_i}, \mathbf{n_j}, ...)$
- **anti dependence**$(\mathbf{n_i}, \mathbf{n_j}, ...)$
- **output dependence**$(\mathbf{n_i}, \mathbf{n_j}, ...)$

のように表す．それぞれの括弧の中は，依存関係が成立する複数のループの展開数を示しており，展開数を外側のループから順番に左から右へ並べるものとする．たとえば，**flow dependence**$(\mathbf{n_i}, \mathbf{n_j})$ は，二重ループの外側を $\mathbf{n_i}$ 回，内側を $\mathbf{n_j}$ 回展開したときにフロー依存関係が出現することをを示している.

3.1.2 繰返し間再利用情報の検出

ループの実行時においては，ある繰返しで生成された値が，それ以降の繰返しにおいて再利用できる場合がある．この関係を繰返し間再利用と呼ぶ.

繰返し間の再利用を検出する場合は，依存検出の場合と異なり，そのすべてを網羅しないと，実行時の結果が異なってしまうということはない．ただし，有用な再利用関係があるときは，これを的確に検出して利用すれば実行の高速化につながる.

本論文では，有用な再利用関係は，

(1) 配列名が互いに同じ
(2) 各添字の式が定数分だけ異なる

という条件を満たすときに成立すると仮定している．この条件は，配列参照の添字多項式を比較することで判別できる.

繰返し間再利用の表記については，

- **reuse**$(\mathbf{n_i}, \mathbf{n_j}, ...)$

のように表す．括弧の中は，再利用関係が成立する複数のループの展開数を示しており，展開数を外側のループから順番に左から右へ並べるものとする．たとえば，**reuse**$(\mathbf{n_i}, \mathbf{n_j})$ は，二重ループの外側を $\mathbf{n_i}$ 回，内側を $\mathbf{n_j}$ 回展開したときに再利用が行えることを示している.

3.1.3 依存および再利用情報とプログラム最適化

以上に述べた依存関係と再利用関係の情報と，2.2 節で述べた最適化の関連を明らかにし，上記の情報がどのように利用されるかについて述べる.

一般に，互いに依存関係のない命令は並列に実行することができるので，フロー依存，逆依存，出力依存の情報をもとに命令間の先行関係を調べ，展開後のループ本体においてどのような並列化が行われるかを把握することができる．また，再利用情報をもとに，値の再利用がどのように行われるかが把握できる．さらに，フロー依存の情報を用いることで，配列変数を介したデータの授受をスカラ変数を介したものに置き換える最適化（スカラリプレースメント）を把握できる．また，出力依存の情報より，どのような命令が省略可能かを把握できる.

3.1.4 具 体 例

図 6 は，図 2 について上記の解析を行った結果である．図 6 では，**i** 方向に 1 回展開したときに，3 個の再利用関係が成立し，**j** 方向に 1 回展開したときに，2 個の依存関係が成立することが分かる.

3.2 展開パラメータの算出

次に，繰返し間依存情報から得られる，ある方向に 1 回展開したときのプログラム依存グラフの最長パスの延びと，再利用情報などから得られる，ある方向に 1 回展開したときのループ本体中の命令減少数を算出する.

ある方向に 1 回展開したときの最長パスの延びは，各依存関係が成立する展開数の組のうち，その方向の成分が 0 以外のものを調べることによって求める．具

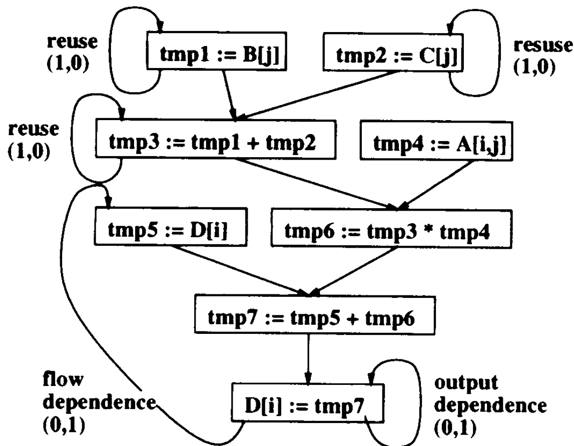

図6 依存グラフと検出情報
Fig. 6 Dependence graph and information derived from the graph.

体的には，ある組について，依存関係が発生する回数だけ展開を行ったときの最長パスの延びを計算し，1回の展開あたりの平均値を求め，その平均値の最大値をとることで算出できる．

ある方向に1回展開したときのループ本体中の命令減少数は，再利用関係が成立する展開数の組のうち，その方向の成分が0以外のものと，フロー依存と出力依存のうち，その方向の成分が0以外で，スカラリプレースメントあるいは命令省略が可能なものを調べることによって求める．具体的には，命令省略およびスカラリプレースメントによる命令減少数は，ある組について，依存関係が発生する回数回展開を行ったときの命令減少数を計算し，1回の展開あたりの平均値を求め，その平均値の和をとることで算出できる．また，再利用による命令減少数は，組の個数により算出できる．これは，ループが十分繰り返されて，再利用が定常状態になったときの値である．

図2の例では，図6の解析より，**j** 方向に1回展開したときに，2個の命令減少，1単位の最長パスの延びがあり，**i** 方向に1回展開したときに，3個の命令減少，0単位の最長パスの延びがあることが分かる．

図7，**図8** に，図2を **j** 方向，**i** 方向へ展開したときの様子を示す．図中太枠で囲い，網かけを行った命令は，展開されて生成された命令を表すものとした．

4. 本手法の概要

4.1 ユニモジュラ変換

多重ループを展開，またはブロック化して実行性能を高めるには，互いに交換可能であるループができるだけ多いほうが有利である．そのため，実際の多重ループ展開に先だって，多重ループの繰返し空間における依存関係の情報を利用し，ループのユニモジュラ変換[6]を組み合わせて適用することにより，全可換な多重ループのネスト数を最大にする．

4.2 ループ展開

多重ループをなす各ループの展開回数は，繰返し間の依存および再利用の関係から，実行時間短縮効果が最大になるような方向を選び，十分な効果が得られる分だけ展開する．ここで，展開回数の見積りにあたって，多重ループが全可換であるか否か，すなわち，最内ループだけでなく外側のループも同時に展開可能かどうかにより，適用される展開手法を以下の2つに分ける．

4.2.1 最内ループのみを展開する場合

多重ループを展開する方向が1方向に限られている場合である．プログラムによっては，繰返し間の依存とループ本体内の依存が循環していることがある．このとき，展開されたループの実行効率は一次的に変化し，その循環に含まれる依存の数，ループ本体の命令数および対象計算機の命令実行能力の関係によって，展開回数を増やせば計算機の並列度までプログラムの並列度を高められる場合と，そうでない場合の両方が存在することが知られている[4]．ここでは，展開回数を決定するにあたって，文献4)により提案された手法を用いる．

4.2.2 複数ループを同時展開する場合

多重ループを展開できる方向が2方向以上ある場合である．この場合の利点は，繰返しの回数が十分多いときに，2方向以上にわたって十分多くループを展開すれば，必ず計算機の並列度までプログラムの並列度を高められることである．この場合の展開回数は，6章で述べるアルゴリズムによって決定される．このアルゴリズムでは，ループ実行性能の見積りに基づいた評価法が利用されているが，これについては5章で詳述する．

5. 多次元展開とループ実行性能の見積り

実行性能を決定する要因には，さまざまなものが考得られるが，我々は，ループの繰返し間の依存関係と生成された値の再利用を，最も重要な要因として考慮している．本章では，これらの情報を利用した実行性能の見積り方法について述べる．

一般には，各ループの展開回数が多いほどループの実行性能が向上するが，その効果の延びは，展開後のループ本体の命令数が多くなりすぎて計算機の命令並列実行能力を超えるころから低下を始める．さらに，レジスタの個数および命令キャッシュの容量などの制

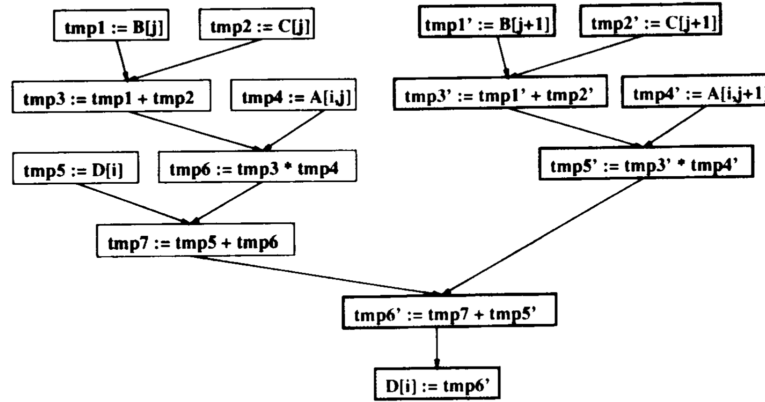

図 7　図 2 の j 方向への展開
Fig. 7　Unrolling loop with index j of Fig.2.

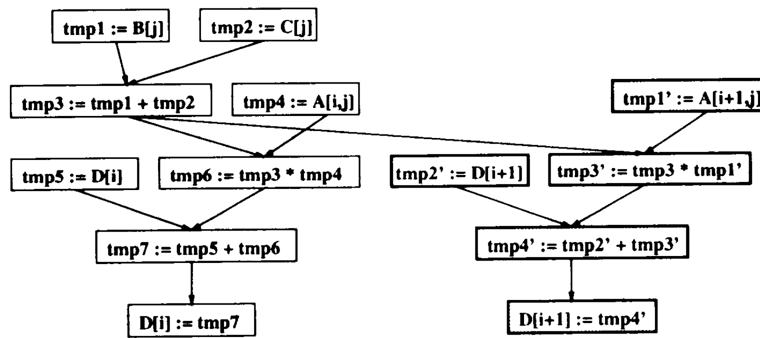

図 8　図 2 の i 方向への展開
Fig. 8　Unrolling loop with index i of Fig.2.

約があるので，展開数をむやみに大きくしても実行性能は向上しない．すなわち，適当な大きさで展開を打ち切ることが重要である．

一般に，並列度の高いプロセッサで実行すればプログラムの実行サイクル（ハードウェアサイクル）は短くなるが，どんなに並列度の高いプロセッサで実行しても，これ以上短いサイクルでは実行ができない下限のサイクル数が存在する（ソフトウェア的な限界）．また，プログラム中の並列度が高ければ高いほどその実行サイクルは短くなるが，どんなに並列度の高いプログラムでも，これ以上短いサイクルでは実行ができない下限のサイクル数が存在する（ハードウェア的な限界）．前者は，依存グラフの最長パスの実行に必要なサイクル数であり，ループ本体の命令間依存関係により決定される．後者は，展開されたループ本体の命令がすべて並列実行されると仮定したときに必要なサイクル数であり，展開後の命令総数，マシンの並列度，および次の命令が実行されるまでのサイクル数により決定される．ループを展開していくにつれ，後者が前者を上回るようになるとループ実行効率の飽和が起きて，実際の展開ループ本体の実行に要するサイクル数が延長する．

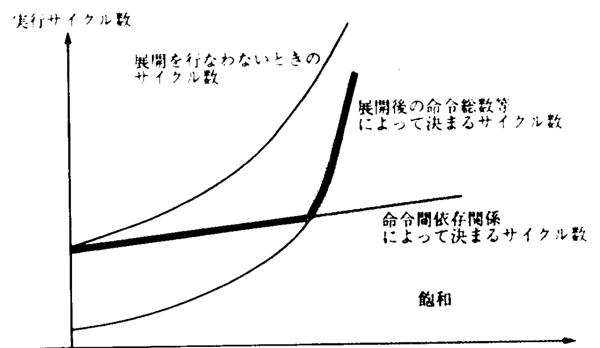

図 9　ループ実行効率の飽和
Fig. 9　Saturation of efficiency in loop execution.

図 9 は飽和の様子を表したものである．図中，展開距離は，各ループの展開数を要素とするベクトルのノルムであり，太線は，ループ展開後の実行サイクルである．後述の式で表すように，展開するループ数を n としたとき，ネストしている複数のループを同時に展開した場合は，展開ループ本体の命令総数によって決まるサイクル数の増加が n 次関数的になるのに対し，命令間依存関係によって決まるサイクル数の伸びは 1 次的である．ループ展開の効果は，展開を行わな

いときのサイクル数と展開後の実行サイクル数の差である．この効果の増加率は，飽和が起こった後に急激に減少する．

最適な展開数を求めるためには，コンパイル時に得られる多重ループに関するパラメタから，展開後の実行性能つまり1サイクルあたりのもとのループの繰返し実行回数，および，ループ実行効率が飽和するかどうかを，多重ループの展開に先だって見積る必要がある．

展開の対象は，ユニモジュラ変換によって得られる全可換な多重ループである．展開対象ループに，外側から順に 1, 2, 3,... と番号を付ける．各ループの展開回数 k_i（展開しないときは1とする）を決めるにあたって，以下のパラメタを考慮する．また，各展開数を要素とするベクトルを $\vec{k}$ で表す．

$m\,(>0)$　1サイクルに実行できる命令の個数
$p\,(>0)$　次の命令が実行されるまでのサイクル数
$N\,(>0)$　展開前のループ本体の命令の個数
$C\,(\geq c_i)$　展開前の最長パスのサイクル数
$c_i\,(\geq 0)$　ループ i を1回展開したときの最長パスの延びにともなうサイクル数の増加
$n_i\,(\geq 0)$　ループ i を1回展開したときに減る命令数

このうち，m と p が計算機アーキテクチャによって決まる定数で，他はすべてプログラムの性質によって決まるものである．これらの値は，前述のようにプログラムの依存・再利用情報の解析で得ることができる．

このとき，以下の指標と関数を得る．

- 命令間依存関係によって決まるサイクル数

$$C_p(\vec{k}) \equiv C + \sum_i \{c_i(k_i - 1)\}$$

- 展開後の命令総数などによって決まるサイクル数

$$C_h(\vec{k}) \equiv \left(N\prod_i k_i - \sum_i \left(n_i(k_i-1)\prod_{j\neq i} k_j\right)\right)p/m$$

- 飽和条件（真のとき飽和）

$$S(\vec{k}) \equiv \left(C_h(\vec{k}) > C_p(\vec{k})\right)$$

- 1サイクルあたりの繰返し回数（ループ実行性能）

$$P(\vec{k}) \equiv \frac{\prod_i k_i}{\max\left(C_p(\vec{k}), C_h(\vec{k})\right)}$$

値の再利用がないとき，つまり，各 i について $n_i = 0$ のときは，性能 P は，飽和が起こった時点で最大になり，その後は $P = m/(pN)$ の一定値をとる．したがって，S が真である点はすべて最適解になる．しかし，ループ展開にともなう値の再利用にともなって，展開後の命令数は飽和後でも減少する．このため，性能は，飽和した場合でも k_i の値の増加にともない緩慢に増大し，$\forall i(k_i \to \infty)$ の極限で

$$\frac{m}{p(N - \sum_i n_i)}$$

に近付く．もちろん，これはループ本体の命令数が無限大の極限において成り立つので，現実的には極限値より低い性能を与える $\{k_i\}$ の値を選ばなくてはならない．しかし，飽和した時点での性能は極限値と大差ないので，性能に関しては飽和領域全体を最適であると我々は考える．

一般に，展開回数 k_i の値が増加したとき，飽和領域の性能が増加する割合は，非飽和領域のそれにくらべて著しく低いので，飽和領域の境界付近を展開点に選ぶのが賢明である．しかし，飽和領域の境界点（面）の性能は，どれもが最適であると判断しているが，それぞれの展開数は異なっており，最も展開数の少ない点を選び，展開後の命令数を少なくすることが必要である．最終的な展開回数と方向は，次章で述べるヒューリスティクスアルゴリズムで決定する．

6. 展開回数の決定アルゴリズム

展開回数の組み $\{k_i\}$ を決定する一手法を紹介する．k_i の値は，1以上の整数に限られるので，まずすべての i について $k_i = 1$ から出発し，i の値を適当に選んで k_i の値を1つずつ増していき，飽和領域に入ったら終了する．このとき，性能がより大きくなる方向を選んで k_i の値を増やすこととする．このアルゴリズムを以下に示す．また，コードサイズを押えるため，展開後の命令数が命令キャッシュ容量を越えた時点でもアルゴリズムを終了する．

(1)　すべての i について $k_i = 1$ に初期化する
(2)　このときの $\{k_i\}$ について S を計算し，S が真ならば終了する．
(3)　展開されたループの命令数が命令キャッシュ容量を越えていれば終了する．
(4)　すべての i について
$$P_i \equiv P(\{k_1, .., k_i + 1, .., k_n\})$$
を計算し，P_i の最大値を与える i を選び，k_i の値を1増す．
(5)　(2)に戻る．

5章で述べた，ループ実行性能 P の式より，P は飽和をむかえるまでは，各方向に単調増加する．したがって，S が真の点を見つけるまでに，局所的に性能の高い点を選んでしまうような方向をとることはありえない．また，飽和領域の境界点のどの点を選択する

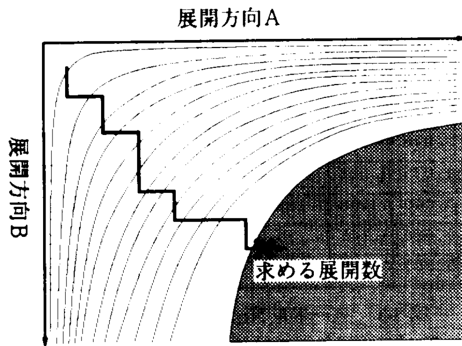

図10 展開回数・方向の決定
Fig. 10 Decision of numbers of times of unrolling and directions.

かは，つねに性能の延びが高い方向を選択し，原点から最も近い点を選ぶことで解決している．この結果，境界点の中で最も展開数が少ない点を選択することが可能である．このアルゴリズムにより展開回数の見積りが進展する様子を**図10**に示す．同図における細線は，二次元のループ多重展開におけるループ実行性能の見積りの等高線で，各軸はそれぞれのループ展開回数である．また，網かけの部分は，ループ実行効率が飽和していることを示すものとした．

7. 評 価

評価の対象として，並列計算機用のベンチマークプログラムであるリバモアフォートランカーネル (Livermore Fortran Kernels)[10] を使用した．カーネル中，多重ループの構造を持つNo.4，8，18，21，23のカーネルを例にとり，VLIW 型並列計算機を対象に最適展開回数を見積り，そのとき得られるループの実行性能を評価した．ここで，ループの実行性能とは，ループ本体コード1サイクルあたりのループ本体中の繰返し数である．**表1**に，見積った展開数およびそのときに実際に得られた相対実行性能を示す．各実行性能は，ループ展開をまったく行わなかったときのループ実行性能を1としたときの相対値であり，各展開数は，ループの入れ子の外側から，各ループの展開数をベクトルで表したものである．ここで，一次元展開とは，最内ループのみの展開を表し，最内ループの最適展開数を得ることのできる文献4) の手法を表すものとする．多次元展開は，本手法により，複数のループを同時に展開する手法である．評価の条件は，1サイクルあたり浮動小数点演算，固定小数点演算，ロード/ストア命令，ジャンプを最大4個実行可能な VLIW 型計算機で，1命令のビット数を256とし，浮動小数点演算および浮動小数点ロード/ストアは2サイクル，その他は1サイクルで実行が完了するものとした．また，TORCH[11]，GIFT[12]のような命令の先行評価をハードウェアサポートとして持つものとし，レジスタの個数については，十分な数を持つものと仮定した．また，命令キャッシュに関しては，256キロバイトの容量を持つものとした．

表1 各カーネルの展開回数および相対実行性能
Table 1 Number of times of unrolling and relative executional efficiency for each kernel.

No.	一次元展開 展開数	一次元展開 相対性能	多次元展開 展開数	多次元展開 相対性能
4	{1,5}	5.00	{3,2}	6.67
8	{1,1}	1.00	{1,1}	1.00
18	{1,2}	1.43	{1,2}	1.43
21	{1,1,6}	5.41	{3,2,2}	7.41
23	{1,1}	1.00	{5,5}	2.99

表1から，各カーネルとも一次元展開したときの実行性能よりも多次元展開したときの実行性能のほうが高くなっていることが分かる．これは，一般的に多次元展開を行うと，一次元展開の場合に比べて，命令の再利用の効果がより強く現れるためである．特に，カーネル No.21 および23においては，一次元展開に比べ，多次元展開のときの実行性能の向上が著しい．カーネル No.21 では，最内ループとその外側のループを2回，さらにその外側のループを3回展開しており，カーネル No.23 では，最外ループと最内ループを5回ずつ展開している．これらのプログラムでは，各ループに繰返し間の依存関係があるため，1方向のみのループの展開では取り出せる並列度に限度がある．多方向を同時に展開した場合は，展開ループ本体の命令数の増加が多次元的になるのに対し，依存サイクル数の伸びは一次元的にしかならない．プログラムの並列度は，1サイクルに実行される平均命令数で決定される．したがって，多方向を同時に展開することで，より大きな並列度を引き出すことが可能である．

次に，展開方向と回数の決定方法の妥当性について述べる．**図11**，**図12**にカーネル No.21 を展開したときの実行性能の見積りと実測値，**図13**，**図14**にカーネル No.23 を展開したときの実行性能の見積りと実測値を示す．単位は展開された繰返し数/サイクル数である．図中，太枠で囲った値は最適と判断される値であり，網かけの部分は実行性能が飽和した領域である．

まず，回数決定方法の妥当性について述べる．図12，図14より，飽和が起こった後では，展開数の増加に対する実行性能の延びは少ないことがわかる．各図の隣合う点間の性能の差分の変化を調べると，図12では非飽和領域での隣あった点間の性能の差の比の平均

		k1							
k3	k2	1	2	3	4	5	6	7	8
1	1	0.143	0.222	0.273	0.308	0.333	0.353	0.368	0.381
1	2	0.286	0.444	0.545	0.615	0.667	0.706	0.737	0.762
2	2	0.571	0.889	1.091	1.143	1.176	1.200	1.217	1.231
2	3	0.828	1.043	1.143	1.200	1.237	1.263	1.282	1.297
3	3	0.857	1.091	1.200	1.263	1.304	1.333	1.355	1.371
3	4	0.873	1.116	1.231	1.297	1.341	1.371	1.394	1.412
4	4	0.889	1.143	1.263	1.333	1.379	1.412	1.436	1.455
4	5	0.899	1.159	1.283	1.356	1.404	1.437	1.462	1.481
5	5	0.909	1.176	1.304	1.379	1.429	1.463	1.489	1.509

図 11 カーネル No.21 の実行性能見積り
Fig. 11 Estimated executional efficiency of kernel No.21.

		k1							
k3	k2	1	2	3	4	5	6	7	8
1	1	0.143	0.222	0.300	0.364	0.417	0.462	0.500	0.533
1	2	0.286	0.444	0.545	0.667	0.714	0.750	0.778	0.842
2	2	0.571	0.800	0.923	1.000	1.053	1.091	1.120	1.143
2	3	0.667	0.923	1.059	1.143	1.154	1.200	1.235	1.263
3	3	0.818	1.059	1.174	1.241	1.286	1.317	1.340	1.358
3	4	0.857	1.091	1.200	1.297	1.333	1.358	1.377	1.412
4	4	0.889	1.143	1.263	1.333	1.379	1.412	1.436	1.455
4	5	0.870	1.143	1.277	1.356	1.389	1.429	1.458	1.481
5	5	0.893	1.163	1.293	1.370	1.420	1.456	1.483	1.504

図 12 カーネル No.21 の実行性能（実測値）
Fig. 12 Measured executional efficiency of kernel No.21.

は 1.800 であり，飽和領域では 1.028 であった．また，図 14 では，同様に，非飽和領域では 1.175 であり，飽和領域では 0.6696 であった．結果として，非飽和領域の性能の差の比の平均と飽和領域の性能の差の比の平均は，図 12 では 1.751 倍，図 14 では 1.754 倍の差があり，各領域で性能の差の延びに関して明白な違いがあることが分かった．したがって，飽和領域の境界で展開を行うのが最も効果的であり，この点の性能を最適値とすることが合理的である．飽和領域の境界を求めるため，本アルゴリズムでは展開空間を探索し，実行性能の飽和が起こった点で探索を終了する．すなわち，本手法は，最適値を求めるための必要条件を満たしているといえる．図 12 では，点 $\{k_1,k_2,k_3\}=\{2,2,2\}$ から点 $\{3,2,2\}$ へ探索点を移動した時点で飽和が発生し，最終的な展開方向と回数を得ている．

次に，方向決定方法の妥当性について述べる．我々は，飽和領域の境界点（面）の性能は，どれもが最適であると判断しているが，その任意の点を選べばよいというわけではない．たとえば，図 14 の $\{4,8\}$ の点は，性能は本手法が算出した位置のものよりも 1.03 倍ほど高いが，展開数が約 1.3 倍になっている．つまり，飽和点の中でも展開数はそれぞれ異なっているの

k1		1	2	3	4	5	6	7	8
	1	0.067	0.074	0.077	0.078	0.079	0.080	0.080	0.081
	2	0.074	0.103	0.118	0.127	0.133	0.138	0.141	0.144
	3	0.077	0.118	0.143	0.160	0.172	0.182	0.189	0.195
	4	0.078	0.127	0.160	0.184	0.202	0.212	0.213	0.213
k2	5	0.079	0.133	0.172	0.202	0.213	0.214	0.214	0.214
	6	0.080	0.138	0.182	0.212	0.214	0.214	0.215	0.215
	7	0.080	0.141	0.189	0.213	0.214	0.215	0.215	0.216
	8	0.081	0.144	0.195	0.213	0.214	0.215	0.216	0.216

図 13 カーネル No.23 の実行性能見積り
Fig. 13 Estimated executional efficiency of kernel No.23.

k1		1	2	3	4	5	6	7	8
	1	0.067	0.080	0.086	0.089	0.091	0.092	0.093	0.094
	2	0.080	0.114	0.133	0.146	0.154	0.160	0.165	0.168
	3	0.086	0.133	0.163	0.177	0.183	0.190	0.194	0.199
	4	0.089	0.146	0.177	0.186	0.194	0.199	0.203	0.205
k2	5	0.091	0.154	0.183	0.194	0.200	0.204	0.207	0.211
	6	0.092	0.160	0.190	0.199	0.204	0.208	0.211	0.213
	7	0.093	0.165	0.194	0.203	0.207	0.211	0.214	0.215
	8	0.094	0.168	0.199	0.205	0.211	0.213	0.215	0.216

図 14 カーネル No.23 の実行性能（実測値）
Fig. 14 Measured executional efficiency of kernel No.23.

で，最も展開数の少ない点を選び，展開後の命令数を少なくすることが必要である．本アルゴリズムでは，展開空間の各点において最も実行性能の見積りの延びが高い方向を選んでいく手法をとっている．この性能の延びの単調増加性は保証されているので，結果的に最適な方向と展開数を算出することができる．この単調増加性は，あくまで 5 章で提示した関数上でいえることではあるが，実際において「最も展開数の少ない点」を探し出すための指標となる．たとえば，図 11 の最適点の決定までの過程では，図 13 との比較より，各点で正しい方向を選択しており，本手法の妥当性を実証しているといえる．また，図 11，図 12 の比較，図 13，図 14 の比較から分かるように，各点における各方向に対する性能の延びの高低は，見積りのものと実測値のものの差が非常に少なく，約 93%の点で正解が得られている．したがって，この見積りにおいて，各点における展開方向の選択を行うことが有効であることが判明した．

表 2 は，各カーネルに対する，本手法における基準上での最適展開数とその見積りであり，本手法の見積りの正しさを実証している．カーネル No.8 および No.18 のような最内ループの展開が最適な場合も，6 章で述べたアルゴリズムは正しく見積りを出しており，見積り方法の正当性がうかがえる．

表 3 は，各カーネルの展開後のループ本体中の命令数である．6 章で示したアルゴリズムでは，展開後の

表2 最適展開回数と展開数の見積り
Table 2 Optimal number of times and estimated ones.

No.	最適展開数	見積った展開数
4	(3,2)	(3,2)
8	(1,1)	(1,1)
18	(1,2)	(1,2)
21	(3,2,2),(2,3,2)	(3,2,2)
23	(5,5)	(5,5)

表3 各カーネルを展開したときの命令数
Table 3 Number of instructions of each unrolloed kernel.

No.	展開数	命令数
4	(3,2)	18
8	(1,1)	54
18	(1,2)	83
21	(3,2,2)	36
23	(5,5)	380

命令数が命令キャッシュを越えることのないように考慮されている．表3より，各カーネルは，それぞれ命令キャッシュの容量を越えることなく展開が行われている．結果として，本手法では，命令キャッシュの容量を越えるようなループ展開は防止されており，ループ展開の効果を落してしまうようなことがないことが分かる．

8. 従来研究との関連

多重ループに関する最適化としては，データの局所性を高めるためのループタイリング，ループブロッキング[13]などがあげられる．論文8)，9)では，ループのユニモジュラ変換[6]を利用して，単純なループブロッキングでは得ることのできなかったデータの局所性を得る方法が述べられている．論文8)，9)においては，最適なループブロッキングのブロック数やユニモジュラ変換の適用法が紹介されており，これを求めるための多重ループの表現や，命令間の依存関係と再利用の表現の形式化がなされている．命令レベルでの並列処理を考慮した場合は，ループ展開を行うことで，データの局所性をさらに高め，生成された値を再利用することが可能になる．これら従来の研究では，ループをブロック化してデータの局所性を高め，キャッシュなどを効率良く利用することが可能であるが，値の再利用をどのように効率的に適用するかは考察されていなかった．我々は，ループブロッキングと多次元ループ展開の関連性から，再利用情報の表現・検出技法を活用し，ループ展開における値の再利用に結び付ける試みを行っている．

9. まとめ

本論文では，多重ループに対する展開の手法ならびにその性質，特に共通化による命令数削減の効果と命令間の依存関係の関係から展開後のループ実行効率の飽和の関係を説明し，各ループの展開回数を決定するアルゴリズムを提案した．この手法により，従来の最内ループのみの展開手法に比べ，2重以上の全可換ループを多重展開することで，命令レベル並列計算機の実行能力をさらに引き出すことが可能であることが判明した．今後は，ループ本体に関する繰返し間の依存および再利用の関係をさらに分析し，収束の速いアルゴリズムに拡張する予定である．また，コードスケジューリング，レジスタアロケーション等との関連からこれらの効果を考慮したアルゴリズムを開発する必要がある．

参考文献

1) Weiss, S. and Smith, J.E.: A Study of Scalar Compilation Techniques for Pipelined Supercompuers, *Proc. 2nd International Conference on Architectural Support for Programing Languages and Operating Systems*, pp.105–109 (1987).
2) Goodman, J.R. and Hsu, W.-C.: Code Scheduling and Register Allocation in Large Basic Blocks, *Proc. 1988 International Conference on Supercomputing*, pp.442–452 (1988).
3) Su, B., Ding, S. and Xia, J.: URPR - An Extension of URCR for Software Pipelining, *Proc. MICRO-19*, pp.104–108 (1986).
4) 諸角，中谷，小松，深澤，門倉：命令レベル並列計算機用最適化コンパイラにおけるループアンローリング技法，信学技報，COMP92-19, pp.23–30 (1992).
5) 細見，森，富田：スーパースカラ・プロセッサにおけるループ最適化，信学技報，CPSY92-32, pp.95–102 (1992).
6) Wolf, M.E. and Lam, M.S.: A Loop Transformation Theory and an Algorithm to Maximize Parallelism, *IEEE Trans. Parallel and Distributed Systems*, Vol.2, No.4, pp.452–471 (1991).
7) Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The Program Dependence Graph and Its Use in Optimization, *ACM Trans. Programming Languages and Systems*, Vol.9, No.3, pp.319–349 (1987).
8) Wolf, M.E. and Lam, M.S.: A Data Locality Optimizing Algorithm, *Proc. ACM SIGPLAN '91 Conference on Programming Language De-*

sign and Implementation, pp.30–44 (1991).

9) Amarasinghe, S.P. and Lam, M.S.: Communication Optimization and Code Generation for Distributed Memory Machines, *Proc. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp.126–138 (1993).

10) McMahon, F.H.: The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range, UCRL–53745, Lawrence Livermore National Laboratory (1986).

11) Smith, M.D., Lam, M.S. and Horowitz, M.A.: Boosting Beyond Static Scheduring in a Superscalar Processor, *Proc. 17th Annual International Symposium on Computer Architecture*, pp.344–354 (1987).

12) 小松，古関，鈴木，深澤：拡張 VLIW プロセッサ GIFT における命令レベル並列処理機構，情報処理学会論文誌，Vol.34, No.12, pp.2599–2610 (1993).

13) Lam, M.S., Rothberg, E.E. and Wolf, M.E.: The Cache Performance and Optimizations of Blocked Algorithms, *Proc. 4th Internaional Conference on Architectual Support for Programming Languages and Operating Systems*, pp.63–74 (1991).

(平成7年12月25日受付)
(平成8年4月12日採録)

古関　聰

1969年生．1992年早稲田大学理工学部電気工学科卒業．1994年同大大学院理工学研究科修士課程修了．現在，同大大学院博士後期課程在学中．アーキテクチャ，コンパイラの研究に従事．

小松　秀昭（正会員）

1960年生．1985年早稲田大学大学院理工学研究科電気工学専攻修了．同年日本IBM東京基礎研究所入社．コンパイラ，アーキテクチャの研究に従事．

深澤　良彰（正会員）

1976年早稲田大学理工学部電気工学科卒業．1983年同大大学院博士課程中退．同年相模工業大学工学部情報工学科専任講師．1987年早稲田大学理工学部助教授．1992年同教授．工学博士．ソフトウェア工学，コンピュータアーキテクチャなどの研究に従事．電子情報通信学会，ソフトウェア科学会，IEEE，ACM 各会員．