システムソフトウェアと
オペレーティング・システム 67－26
（1994. 12. 9）

多重ループにおける最適ループ展開数算定技法

古関 聰 † 今野 明 † 小松 秀昭 ‡ 深澤 良彰 †
† 早稲田大学理工学部 ‡ 日本 IBM(株) 東京基礎研究所
新宿区大久保 3-4-1 早稲田大学理工学部

あらまし

プログラム実行においてループの占める割合は高く、ループアンローリングによって展開されたループを並列化することによるプログラム実行の高速化の効果は高い。従来は最内ループのみの展開を行なっており、多重ループ間の命令共通化、再利用を十分に行なうことができなかった。そこで、本稿ではプログラム中の命令間の依存関係や再利用関係とターゲットアーキテクチャのマシンリソースを考慮することによって最適に近いループ展開数と方向を見積る方法を述べる。また、ベンチマークテストを用いて本手法の効果を評価した。

和文キーワード

A Method For Estimating Optimal Unrolling Numbers for Nested Loops

Akira Koseki† Akira Konno† Hideaki Komatsu‡ Yoshiaki Fukazawa†
† School of Science & Engineering, Waseda University
‡ Tokyo Research Laboratory, IBM Japan, Ltd.
School of Science & Engineering, Waseda University, 3-4-1 Okubo, Sinjuku-ku

Abstract

Loop unrolling is one of the most promising parallelization technique, because programs have a nature that most processing time is spent in their loops. Existing compilers unroll only innermost loop, so they cannot perform commoning subexpresion and resusing instructions. In this papar, we propose a heuristic estimation method, in which some properties of programs, for example, dependency distance, resuse, and so on, are utilized. Our method is evaluated with benchmark tests.

英文 key words

1 はじめに

数値演算プログラムでは、全体の実行時間に占めるループ部分の実行時間の割合が大きいので、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。幸い、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、今までに多くの研究がなされている。

その中に、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り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、効果的な方法である。特に、繰返し間の制御依存が解消され、命令の繰返しを越えた移動が可能になることは、最も重要である。即ち、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、並列度が大幅に高まる可能性がある。また、近隣の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、それらの繰返しを展開して一つにまとめると、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が削減され、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、特に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発表されている[1][2]。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 [3] 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と、繰返し間依存のウェーブフロント方向 [3] の並列度の抽出が可能になったり、値生成を再利用できる範囲がさらに広がる。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方が、より高い実行高速化の効果が期待できる。

しかし、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異なる。展開するループによっては、配列の参照が共通化でき、1回の展開当たりのメモリ参照命令の個数を減らして、ターゲットマシンの実行ユニットを使い切ってしまうまでに、より多くの繰返しを展開できるものもある。したがって、最適なループ展開回数を決定するには、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、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、繰返し空間における依存関係および再利用関係を計算する。

本稿では、以下の順に、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んな効果があるか、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、特に依存ベクトルおよび再利用ベクトルを抽出する方法を説明する。最後に、これらの情報を用いて最適な各ループの展開回数を見積もるアルゴリズムを示し、これによりプログラムの実行性能がどのくらい高まるかを評価する。

2 多次元展開の性質

多次元展開の例を図1に示す。プログラム1のループに対し、iについて2回、jについて3回の繰り返しからなるブロックを構成し、ブロック内の全ての繰返しを展開し、1個の繰返しにまとめたものがプログラム4である。このような変換を行うには、途中でiのブロック内ループとjのブロック制御ループを交換していることから、ループ本体にはiのループとjのループの交換を禁止するような繰返し間の依存関係があってはならないことがわかる。

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

プログラム 1: 対象プログラム

このループ本体に対する依存グラフを図1に示す。

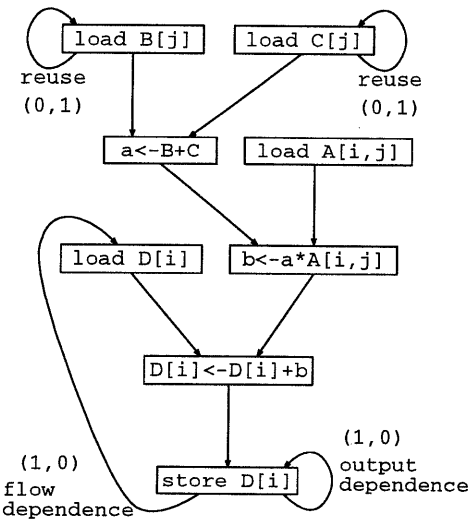

図1 ループ本体の依存グラフ

このループにi,j方向とも展開するために、例えばiについて2回、jについて3回のイタレーションをひとまとめにして、元のループから分離する。これをプログラム2に示す。

```
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]);
```

プログラム2: ループ分割

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、イタレーションの実行順序は変わっていない。この中で、iのループとjjのループを交換し、内側2個に6回の固定したイタレーションからなるループの入れ子、いわゆるブロックを得る。この操作は、元の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イタレーション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。

```
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]);
```

プログラム3: ブロッキング

最後に、ブロックをなす入れ子ループ内の全てのイタレーションを展開し、1つのループ本体にする。これ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、合計6個のイタレーションからなる新しいループ本体を作った。

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

プログラム4: 多次元展開

2.1 展開後のループ本体の命令共通化

次のステップとして、ループ間に渡る並列性の抽出と重複する命令の削除を行い、展開されたループ本体の最適化を進める。

まず、配列要素B[jj],B[jj+1],B[jj+2]をロードする命令はそれぞれ2回存在し、しかも常に同じ値を得ることがわかっているので、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、ロード命令を使う代わりに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は、それぞれ1回に節約でき、命令を3個減らせる。これをスカラーリプレースメントという。同様に、配列Cについても3個のロード命令を減らせる。

値生成の共通化(commoning)による命令削減は、ロード命令だけにとどまらず、四則等の演算にも現れることがある。この例では、値B[jj]+C[jj]の生成は、2回出現するが、いずれも同じ値になることがわかっている。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、そのレジスタの値で代えて、1個の加算命令を節約する。同じことが`B[jj+1]+C[jj+1]`,`B[jj+2]+C[jj+2]`についても成り立ち、合計3個の加算命令を減らせる。

配列`A`については命令共通化の効果が無いが、`D`については、配列参照による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[ii]`は、2回目の右辺の`D[ii]`に対してフロー依存をなしているが、2回目の右辺の`D[ii]`の値は1回目の左辺の`D[ii]`の値と常に等しいので、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。同様にして、3回目のロード命令も省略できる。一方、1回目の左辺の`D[ii]`と2回目の左辺の`D[ii]`の間には出力依存が存在し、しかも1回目の左辺の`D[ii]`の値は常に2回目の左辺の`D[ii]`により消される。したがって、2回目の右辺の`D[ii]`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[ii]`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。同様に、2回目のストア命令も省略できる。以上、`D[ii]`に関するメモリ参照命令は6個から2個に減る。これは、`D[ii+1]`についても成り立つ。また、このループ本体は、繰り返し実行されるので、命令減少の効果は大きなものとなる。

ブロッキングの効果として配列によるメモリ参照を局所化し、キャッシュミスを抑える効果のあることが知られているが、ループアンローリングの場合は、命令の発行および実行回数を抑え、計算機の実行時間を短縮する効果となって現れる。したがって、展開するループおよびその回数を決める際には、そのことも十分考慮する必要がある。

3 ループの検出

3.1 CFG作成

ソースプログラムは、まず、フロントエンド処理（字句解析、構文解析、意味解析）を経て中間コードに変換される。中間コードには、ジャンプ命令等の分岐を行う命令と、合流点に関する情報が記載されているので、これを利用して制御流れ図(Control Flow Graph)を作成する。制御流れ図は、基本ブロックと呼ばれる、分岐や合流のない命令の集まりと、それらを結ぶ制御エッジからなる。この段階では、命令間の依存関係はまだ不明である。

3.2 レベル付け

制御流れ図から強連結部分を見つける。制御の流れは、ほとんどの場合簡約可能であるが、簡約不能の構造になっている場合は、最適化を容易にするために、節点分離を行って簡約可能な構造に変更する。これで、分岐構造はループとif型構造の2種類になる。

各基本ブロックに対し、ループの深さを表すレベルを付ける。まず、制御流れ図のうち、強連結部分に含まれない基本ブロックのレベルを0とする。これは、ループに属さないことを示す。これに対し、強連結部分に含まれる基本ブロックのレベルを1とする。もし、さらに別の強連結部分に含まれるならば、レベルを1つ増す。

3.3 制御変数の抽出

ループを1回繰り返すに値が定数分上下するような変数をそのループの制御変数という。もしそのような変数があって、しかもループの終了条件がその変数の値で指定されていれば、その変数および初期値、終了値、増分を記録する。

3.4 依存および再利用情報の抽出

以上の操作により分岐構造はループおよびif型構造の2種類だけとなったが、さらに命令に実行条件を追加して、if型構造を解消し、ループを単一の基本ブロックから構成されるように変換する。さらに、大域的データ流れ解析を行い、PDG(Guarded Program Dependence Graph)[4]を作成する。

3.4.1 繰返し間依存情報の検出

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全て網羅しなければならない。しかし、実行時において依存関係の発生する可能性のないものも取り上げてしまうと、最適化を妨げてしまうことになる。したがって、なるべく正確なデータ流れ解析をしたい。そのためには、配列参照に関する正確な情報を必要とする。そこで、今回の研究では、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。

一方、配列名の同じメモリ参照は、依存および再利用の可能性があるから、各添字の値が繰返しによってどのように変化するかを調べる必要がある。

3.4.2 繰返し間再利用情報の検出

繰返し間の再利用の検出の場合は、依存検出の場合と異なり、必ずしも正確に調べる必要はない。ただ、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。有用でない再利用関係というのは、ループを展開しても、各繰返しではなく、実際には一部の繰返ししか再利用の効果が得られないものである。例えば、図1のループをブロッキングしても、この場合、配列aの再利用は、対角線上 $(i = j)$ にまたがったブロックしか得られない。そのような再利用関係は、利用価値が低いので無視する。

```
for i:=1 to IMAX do
  for j:=1 to JMAX  do
    b[i,j]:=a[i,j]+a[j,i];
```

プログラム 5: 有用でない再利用

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。この条件は、配列参照の添字多項式を比較すれば容易に判別できる。

4 ループの変換

4.1 ユニモジュラー変換

多重ループを展開したりブロッキングをして実行性能を高めるには、レベルを互いに交換可能なループの次元ができるだけ多いほうが有利である。そのため、多重ループの繰返し空間に張るベクトルの情報を利用して、スキューイングの組み合わせにより全可換な多重ループの次元数を最大にする。この変換を求めるためにユニモジュラー変換 [3] を用いる。

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、2つに分ける。

4.2.1 最内ループのみを展開する場合

この場合は、多重ループを展開する方向が1方向に限られている。プログラムによっては、1方向にループを繰り返すことで発生する依存とループ本体内の依存が結合して依存循環が発生していることがある。このとき、その依存循環の長さ、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 [1]。ここでは、展開回数を決定するにあたって [1] 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

この場合は、多重ループを展開できる方向が2方向以上ある。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決定法には、5章で述べる評価法を用いた、6章で述べるアルゴリズムを用いる。

5 多次元展開による実行性能の見積り

一般には各ループの展開回数が多い程全体の並列度が向上するが、その効果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し、更にレジスタの個数および命令キャッシュの容量の制約があるので、むやみに大きくすることはできない。即ち、適当な大きさで打ち切ることが重要である。

1サイクル当たりの命令実行能力は有限のため、ループ本体の命令間依存関係により決まる最小の実行サイクル数（依存サイクル数）C の間に実行可能な命

令の個数に対して、展開ループ本体の命令総数が上回るようになると、命令数の飽和が起きて、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。

そのために、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サイクル当たりの元のループの繰返し実行回数、および、展開ループ本体の命令数が飽和するかどうかを、ループの変換に先だって見積る必要がある。

展開の対象にする最も内側の n 重のループに、外側から順に $1,2,3,\ldots$ と番号を付ける。各ループの展開回数 k_i（展開しない時は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。

$m(>0)$	1サイクルに実行できる命令の個数
$n_0(>0)$	展開前のループ本体の命令の個数
$c_0(\geq c_i)$	展開前の依存サイクル数
$c_i(\geq 0)$	i 番目を1回展開して増えるサイクル数
n_i	i 方向に1回展開した時に減る命令数

m だけ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る。

このとき、以下の値を見積る。

$C \equiv c_0 + \sum_i c_i(k_i - 1)$　展開後の依存サイクル数

$N \equiv n_0 - \sum_i n_i \frac{k_i - 1}{k_i}$
展開後の繰返し1回当たりの命令数

$S \equiv N \prod_i k_i - m/C$　飽和条件 ($S > 0$ のとき飽和)

$P_{NS} \equiv \frac{\prod_i k_i}{C}$　飽和しないときの性能

$P_S \equiv m/N$　飽和したときの性能

変数の再利用がないとき、つまり $n_i = 0$ のときは、性能 P は、命令実行能力が飽和したときに最大になる。また、飽和したときの性能は $P = m/n_0$ で一定になる。したがって、$S = 0$ を満たす点は全て最適解になる。しかし、ループ展開に伴うメモリ参照の共通化に伴って繰返し1個当たりの命令数が若干減るという効果がある。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

$$\frac{m}{n_0 - \sum_i n_i}$$

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、実際には、これより若干低い性能を与える $\{k_i\}$ の値で妥協しなければならない。しかし、もし、そのときの性能が極限値と大差なければ、十分最適であるといえる。

一般に、展開回数 k_i の値が増加したとき、性能が増加する割合は、飽和している場合は飽和していない場合にくらべて低いので、飽和領域の境界付近を選ぶのが賢明である。その場合でも、性能最大の点と命令数最小の点は一般には一致しない。したがって、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から、まず全ての 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 の値を増やせばよい。このアルゴリズムを以下に示す。

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 の値を計算し、$S \geq 0$ ならば終了する
3. すべての i について $P_i \equiv P(k_1,\ldots,k_i+1,\ldots,k_n)$ 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
4. 2. に戻る

5章の P の式より、P は、プロセッサの命令実行能力が飽和するまでは、各方向に単調増加する。従って、$S \geq 0$ の点を見つけるまでに、局所的性能の延びが高いが、大域的には性能が低い点を選ぶような方向をとることはあり得ない。

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2に示す。この図で、影のついた部分は、命令実行能力が飽和していることを示す。プログラムはリバモアカーネルの23番を用い、ターゲットアーキテクチャの並列度を4とした

ki \ kj	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
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
9	0.081	0.146	0.200	0.214	0.215	0.216	0.216	0.217
10	0.081	0.148	0.204	0.214	0.215	0.216	0.216	0.217

図 2: カーネル No.23 の実行性能見積り

7 評価

リバモアカーネルの No.4,8,18,21,23 のプログラムを例にとり、VLIW 型並列計算機を対象に、最適展開回数を見積り、その時得られる性能つまり 1 サイクル当たりの繰返し実行回数を調べた。表 1 に、見積った展開数およびその時に実際に得られた実行性能を示す。ここで、A は最内ループのみを展開する手法によるもの、B は本手法によるものを表す。また、評価の条件は、1 サイクルあたり浮動小数点演算および浮動小数点ロード / ストアを最大 4 個、その他の命令は無制限実行可能な VLIW 型計算機で、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。

表 1 各カーネルに対する展開回数および実行性能

No.	A 展開数	A 性能	B 展開数	B 性能
4	(1,5)	1.000	(3,2)	1.333
8	(1,1)	0.074	(1,1)	0.074
18	(1,2)	0.096	(1,2)	0.096
21	(1,1,6)	0.774	(3,2,2)	1.091
23	(1,1)	0.083	(5,5)	0.213

この表から、各カーネルとも 1 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方が若干高くなっていることがわかる。これは、多次元展開を行うと、1 次元展開の場合に比べて、記憶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.23 においては、多次元展開すると実行性能が格段に向上する。このループには、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。2 方向を同時に展開した場合は、展開ループ本体の増加が 2 次元的になるのに対し、依存サイクル数の伸びは 1 次元的にしかならないので、展開ループ本体の命令数が飽和するまで限りなく並列度を高められる。

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。カーネル No.23 では、最内ループとその外側のループを 5 回づつ展開している。これらを決定するアルゴリズムは、6 章に示してある。このループは、一方向を何度展開しても実行性能はあまり向上せず (図 2)、多数の方向を展開することによってさらなる性能向上が得られる性質を持っている。本アルゴリズムでは、展開空間の各点において最も性能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。図 2 は、その様子をよく表しており、本手法の妥当性を実証していると言える。

8 まとめ

今回の発表では、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実行ユニット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、2 重以上の自由交換可能多重ループから細粒度並列計算機の実行能力を最大限に引き出すループ変化が常に可能であることがわかった。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] 諸角裕, 中谷信太郎, 小松秀昭, 深澤良彰, 門倉敏夫, "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, Vol.92, No.108, pp.23-30(1992).

[2] 細見岳生, 森眞一朗, 富田眞治, "スーパースカラ・プロセッサにおけるループ最適化", SWoPP'92, CPSY92-32, pp.95-102(1992).

[3] M.E.Wolf, and M.S.Lam, "A loop transformation theory and an algorithm to maximize parallelism", IEEE Trans. Parallel and Distributed Systems, Vol.2, No.4, pp452-471(1991).

[4] J.Ferrante, K.J.Ottenstein, and J.D.Warren, "The Program Dependence Graph and Its Use in Optimization", ACM Trans. Programming Languages and Systems, Vol.9, No.3, pp.319-349(1987).

盛光印刷所