命令レベル並列アーキテクチャのためのレジスタ割付け技法

小松秀昭[†] 神力哲夫[††,☆]
古関聰[††] 深澤良彰[††]

スーパスカラ，VLIWといった，命令レベルでの並列実行が可能なアーキテクチャを持つプロセッサでは，レジスタ割付けにおいて，従来は考える必要のなかった新たな問題が発生する．第一に，異なる変数に対して同一のレジスタを割り当てた場合，その命令間に新たに逆依存の関係が発生するが，これによりプログラムの並列性を損なう可能性がある．そこで，発生する逆依存がプログラムのクリティカルパスを不必要に伸ばさないよう留意することが必要である．第二に，プロセッサの並列性を利用すると，スピルコードの挿入が必要となった場合，それらを他の命令と並列に実行することが可能である．そこで，そのような最適な挿入位置を考慮する必要がある．いずれの場合も，プロセッサが命令レベルでの並列性を持たない場合には問題にならないため，従来のレジスタ割付け手法では考慮されていない．そのため，従来のレジスタ割付け手法をそのまま命令レベル並列プロセッサに利用した場合，プログラムの持つ並列性を損ないプロセッサの性能を十分に活用できなくなってしまう．本稿では，プログラムを命令間の依存関係を表すグラフ構造で表現することで，上記の問題を解決するレジスタ割付け手法を提案し，その性能を従来手法と比較する．

A Register Allocation Technique for Instruction-Level Parallel Architecture

HIDEAKI KOMATSU,[†] TETSUO SHINRIKI,[††,☆] AKIRA KOSEKI[††]
and YOSHIAKI FUKAZAWA[††]

On a register allocator for instruction-level parallel processors, such as superscalar, VLIW, there exist some problems we have not needed to consider when we were using scaler processors. First, when the same register is allocated for some different variables, anti-dependence is generated and it may decrease instruction-level parallelism in a program. Secondly, we must think of the best positions where spill instructions are inserted because spill instructions can be parallelly executed with other instructions by taking advantage of parallel execution units of a processor. These problems may not play an important role in scalar processors which do not have parallel execution units, so existing register allocators do not take them into considerations. Therefore, when conventional register allocators are applied to instruction-level parallel processors, their parallelism can not be reflected to an object program. This paper describes the new register allocation algorithm to solve these problems, using the graph structure which represents instructions and dependence between them. This paper also evaluates our new algorithm in comparison with conventional register allocators.

1. はじめに

近年，スーパスカラ，VLIWといった，複数の演算ユニットを持ち命令レベルでの並列実行が可能なプロセッサが，数多く研究開発されており，既に実用化が進んでいるものも数多く存在する．このようなプロセッサにおいては，その性能を引き出すための最適化技術が重要であり，それに関連したさまざまな技術が研究・開発され，高い成果を上げている．

最適化コンパイラで重要な技術の一つにレジスタ割付けがある．通常の命令レベル並列プロセッサでは，複数の演算ユニットが同時に一つのレジスタファイルを読み書きするので，レジスタアクセスに関する最適化が重要である．しかし，現状では，命令レベルでの

† 日本IBM（株）東京基礎研究所
IBM Japan Ltd. Tokyo Research Laboratory
†† 早稲田大学理工学部
School of Science & Engineering, Waseda University
☆ 現在，大日本印刷（株）
Presently with Dai Nippon Printing Co., Ltd.

並列性を考慮せずに開発された従来の割付け手法を，そのまま利用していることが多い．そのため，コードスケジュールによって得られた高い並列性が，その後に行われるレジスタ割付けによって大きく損なわれてしまっている．

本手法では，プログラムを，コードスケジュールでよく用いられるようにグラフ構造によって表現し，その上でレジスタ割付けを行っていく．これにより，プログラムの並列性に注意を払いながら，そのクリティカルパスができる限り伸びないよう，レジスタ割付けを行うことを可能にしている．

2. 本研究の背景

2.1 レジスタ割付けと逆依存

一般的なコンパイラでは，まず，無限個の仮想レジスタを仮定し，その上でコードスケジュールなどの最適化処理を行う．レジスタ割付けでは，この仮想レジスタを限られた個数の物理レジスタへ割り付けていく．そのため，別々の仮想レジスタが，同一の物理レジスタへ割り付けられる場合があり，これによってプログラム中に逆依存が発生する．この逆依存がプログラムへ及ぼす影響について，**図1** (a) へのレジスタ割付けで考えてみる．

同図 (b) はプログラムを命令間の依存関係を表すグラフ構造で表現したものである．命令レベル並列アーキテクチャを対象として考えた場合，プロセッサの並列度さえ十分であれば，グラフ上で前後関係のない命令は並列に実行できるため，このグラフの高さがプログラムの実行時間に対応することになる．したがって，このプログラムの実行には3サイクルが必要である．

図2 は図1に対してレジスタ割付けを行った結果の一つである．変数 t1 と変数 t3 を同じレジスタ r4 に割り付けているため，同図 (b) の点線のような逆依存が発生し，命令1と命令3を同時に実行することができない．そのため，実行に必要なサイクル数が3から4へと増加している．逆に，**図3** のような割付けを行えば，始めから依存関係のある命令1と命令2および命令3と命令4の間に発生した逆依存が隠れるため，並列性を落とすことなく，すなわち，実行時間を伸ばすことなくレジスタ割付けを行うことができる．

命令レベル並列プロセッサでは，発生した逆依存によってプログラムの並列性を落とすことのないよう注意を払い，後者のような割付けを行うことが必要である．しかしながら，並列性のないプロセッサでは，この逆依存の発生は問題にならなかったため，広く用いられているグラフ彩色を利用したレジスタ割付け手法[1),2)]では，このようなことは考慮されていない．

これらの手法では，図1(a) のように並べられた命令列から各変数の生存区間を解析し，生存区間の重なる変数は互いに干渉する（同じレジスタを割り当てられない）ものとして扱っていた．そのため，図1の変数 t1 と変数 t3 は互いに干渉しないと判断し，これらの変数に同一のレジスタを割り付けてしまう可能性がある．この場合，図2のような割付けを行う可能性がある．

そこで，本手法では図1(b) のようなグラフ構造上でレジスタ割付けを行い，命令の並列実行を保持するために，変数 t1 と変数 t3 は干渉すると考えることにする．すなわち，これまでの手法では生存区間の重なる変数のみが干渉していると考えたが，それに加えて並列実行できる命令間の変数間も互いに干渉すると考

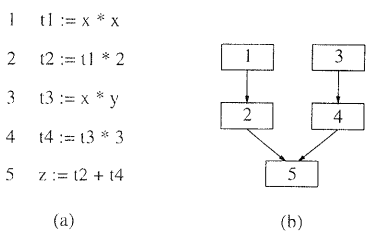


図1 並列プログラムの例
Fig. 1 Example of parallel program.

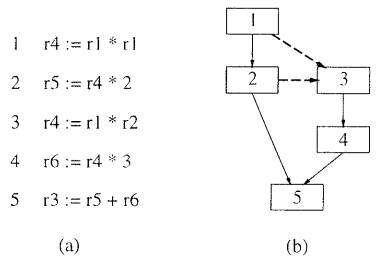


図2 並列性を失う割付け
Fig. 2 Register allocation which loses parallelism in a program.

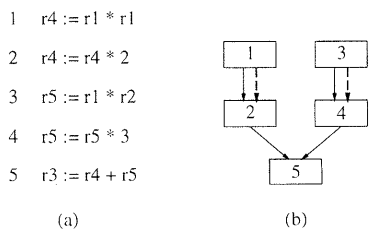


図3 並列性を保持する割付け
Fig. 3 Register allocation which keeps parallelism in a program.

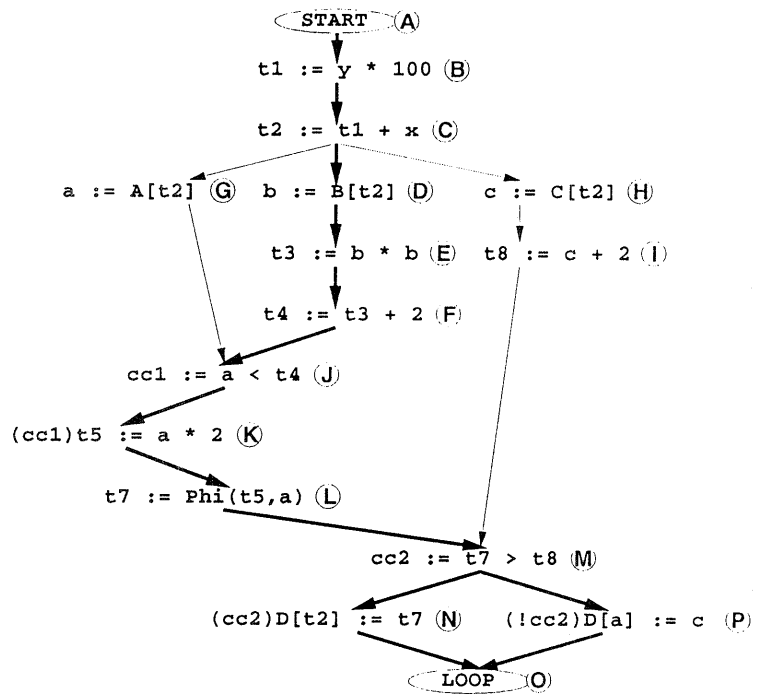


図4 GPDGの例
Fig. 4 Example of GPDG.

えることで，その命令の並列実行を保証する．

2.2 スピルコードの並列処理

プログラム中で使用される変数が多い場合，すべての変数をそのままレジスタに直接割り付けることはできない．この場合，一つのレジスタを複数の変数で共有する必要がある．そのためには，計算された変数の値をメモリ上に書き出したり，逆に，書き出しておいた値をレジスタへ読み戻したりすればよい．このような目的でレジスタ割付けフェーズがプログラム中に挿入する命令を，一般にスピルコードと呼ぶ．

プロセッサが命令レベルの並列性を持たない場合，このような命令が挿入されることはプログラムの実行時間を伸ばすことにつながる．そのため，従来のレジスタ割付けでは，このスピルコードの挿入を最少化することが重要視されてきた．

しかしながら，命令レベル並列プロセッサでは，その並列性を利用して，スピルコードを他の命令と並列に実行することが可能なため，スピルコードの最少化は大きな意味を持たなくなる．逆に，それ以上にスピルコードの並列実行を考慮することが必要となってくる．

3. 本手法の概要

3.1 コンパイラの構成

本手法が想定するコンパイラは，プログラムをループ構造を単位として階層的に分割し，内側のループから外側のループへとコードスケジュールおよびレジスタ割付けを行っていく．このようなループ構造を利用した階層的なレジスタ割付けは，文献3)にも述べられているが，実行時に多くの時間を費やす内側のループほど優先的にレジスタを利用できる利点がある．

また，コンパイラはコードスケジュールおよびレジスタ割付けに先んじてSSA変換[4]を行い，プログラム中の逆依存を排除する．

3.2 GPDG

GPDG (Guarded Program Dependence Graph)[6]とは，PDG[5]を改良したもので，条件分岐構造の表現に特徴がある．図4において，cc1，cc2を条件変数と呼び，ある条件分岐の分岐結果を表す仮想的な変数である．各命令には，条件変数の真偽の組合せによる実行条件が付加され，これによって条件分岐構造を表している．例えば，命令Kは，a < t4が満たされた場合にのみ実行される．

このGPDGはプログラムのループを単位として作

成される．そのため，ループの先頭を表す START ノードから，最後尾を表す LOOP ノードへ至る非循環な有向グラフとなる．

以下に，本稿で必要な GPDG 上のいくつかの言葉の定義を行う．

経路（パス） ノード A から，ノード $B,C,\ldots$ を通ってノード Z へ到達可能なとき，その道筋を“経路”または“パス”と呼び，A-B-C-...-Z と表す．

経路の長さ 経路 P の“長さ” $Length(P)$ を次のように定義する．

$Length(P)$ = 経路 P に含まれるノードの個数

ノードの自由度 ノード X の“自由度” $Free(X)$ を次のように定義する．

$$Free(X) = Depth(\texttt{LOOP}\ \text{ノード}) - (Height(X) + Depth(X)) + 1$$

ただし，ノード間の“距離” $Distance$，ノード“深さ” $Depth$，ノードの“高さ” $Height$ を，それぞれ以下のように定義する．

$$Distance(X,Y) = \mathrm{Max}(Length(X\ \text{から}\ Y\text{への経路}))$$

$$Depth(X) = Distance(\texttt{START}\ \text{ノード}, X)$$

$$Height(X) = Distance(X, \texttt{LOOP}\ \text{ノード})$$

分岐点 ノードが“分岐点”であるとは，そのノードが複数の子ノードを持つことをいう．図 4 では，{C,M} が“分岐点”である．

合流点 ノードが“合流点”であるとは，そのノードが複数の親ノードを持つことをいう．図 4 では，{J,M,O} が“合流点”である．

クリティカルパス START ノードから LOOP ノードへ，自由度が 0 のノードのみを通って至るパス（複数存在する可能性がある）を，クリティカルパスと呼ぶ．図 4 で，太線で表されている経路がクリティカルパスである．

3.3 GPDG とレジスタ割付け

命令レベルの並列性を持ったプロセッサでプログラムを実行する場合，GPDG 上で前後関係のない，すなわち，依存関係のない命令は互いに並行に実行することができる．そのため，プロセッサの並列度が十分であれば，プログラムの実行時間は，GPDG 上の最長のパス，すなわちクリティカルパスの長さによって決定される[7]．したがって，クリティカルパスの長さを，最適化の指標として捉えることができる．レジスタ割付けフェーズでは，このクリティカルパスをできるだけ伸ばさないように，各変数に対してレジスタを割り当てていくことを考えればよい．

3.4 命令の自由度とレジスタ割付け

命令の自由度は，このクリティカルパスを伸ばさずに，その命令の実行を遅延できる最大のサイクル数を表す．例えば，図 4 中の命令 G の自由度は 2 であるが，これは，この命令の実行を 2 サイクル，すなわち，命令 F を実行する時点まで遅らせても，プログラムのクリティカルパスを伸ばさずに済むことを示している．ノードの自由度のスケジューリングへの応用は，文献 7) などでも行われているが，本手法ではこれをレジスタ割付けに応用する．

プログラムに対してレジスタ割付けを進めていくと，プログラム中にスピルコードが挿入され始める．このとき，そのレジスタを利用する命令の実行は，スピルコードの実行を待たなければいけなくなるため，その分だけ遅れてしまう．もし，この命令の自由度が低かったとすると，プログラムのクリティカルパスを大きく伸ばしてしまうことになる．逆に，自由度の低い命令の実行が多少遅れたとしても，クリティカルパスの長さへの影響は小さく留めることができる．

このような理由から，本手法では，自由度の低いノードから順にレジスタ割付けを進めていく．このようにすることで，スピルコードは自由度の高いノードの前に挿入されることになり，クリティカルパスの長さ，すなわち，プログラム全体の実行時間が延びることを抑制する．

4. 本手法の流れ

本手法では，**図 5** のようにレジスタ割付けが行われる．各処理の詳細を以下に述べる．

4.1 レジスタ割付けの前処理

パス分割 パス分割フェーズでは，与えられた GPDG を次のような条件を満たすパスに分割する．

- 各パスは分岐点ノードもしくは START ノードから始まり，合流点ノードもしくは LOOP ノードで終わる．
- 各ノードは，それを含むパスが必ず一つ以上存在する．ただし，分岐点・合流点・START ノードお

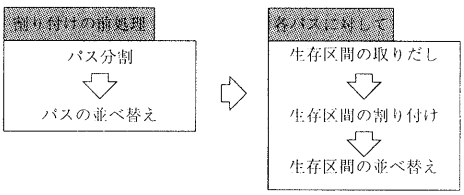


図 5 レジスタ割付けの流れ
Fig. 5 Components of the register allocation.

よびLOOPノード以外のノードは，それを含むパスが複数あってはならない．

- 連続する，自由度の同じノードは，その双方を含むパスが必ず存在する．

この条件にあてはまるよう図4を分割すると，A-B-C-D-E-F-J-K-L-M-N-O，M-P-O，C-G-J，C-H-I-Mの4つに分割される．以降では，これらの経路を順番にPath1，Path2，Path3，Path4と呼ぶことにする．上の定義では分割結果は一意に決まらないが，そのどれであっても構わない．

$X \rightarrow Y$ なる依存関係のあるノード X と Y において，Y が合流点ノードでなければ，以下の式が成り立つ．

$$Depth(Y) = Depth(X) + 1$$

このことは，START ノードから Y への経路は，Y が合流点ノードでないため，X を通り Y に至る経路しか存在しないことから自明である．

同様に，X が分岐点ノードでなければ，以下の式が成り立つ．

$$Height(X) = Height(Y) + 1$$

したがって，X が分岐点ノードでなく，かつ，Y が合流点ノードでなければ，以下の式が導かれる．

$$Free(X) = Free(Y)$$

このことから，各パスに含まれるノードの自由度は，パスの最初と最後の二つのノード以外はすべて同じになることがわかる．以　　は，この自由度を「パスの自由度」と呼ぶ．

パスの並べ替え　このフェーズでは，分割されたパスを自由度の低い順番に並べ替える．自由度が同じパスに関しては，特に順番は指定しない．前述のように，自由度の低い経路中にスピルコードが挿入されることは好ましくない．並べ替えられた順番にレジスタ割付けを行うことにより，自由度の低い経路ほど優先的にレジスタを利用できるようにする．前述の各経路は，Path1-Path2-Path3-Path4の順に並べ替えられる．

4.2 各経路のレジスタ割付け

生存区間の取り出し　切り出された経路中には，複数の変数が含まれている．レジスタ割付けを行うためには，それらを解析し，各変数の生存区間を求める必要がある．例えば，上述のPath1中には11個の変数が含まれており，それぞれの生存区間は**図6**のように解析される．

生存区間の並べ替え　取り出された生存区間を，命令の密度（生存区間中の定義・参照回数 ÷ 生存区間の長さ）に従って並べ替え，密度の高い方から順に割付けを行っていく．これは，レジスタの使用効率を高くするためである．例えば，図6の変数 x の生存区間の場合，1回目の参照から2回目の参照までかなりの時間がある．このような生存区間を他の命令に先駆けてレジスタに割り付けた場合，そのレジスタが演算に使われることはなく，後々の演算に備えて計算された値を保持するためだけに，レジスタを占有してしまうことになる．もし，この変数 x がレジスタを占有したことで，b，t3，t4，a，t7 のような密度の高い生存区間の割付けが不可能になったとすると，非常に多くのスピルコードが挿入されてしまうことになる．しかしながら，密度の低い生存区間 x が割り付けられなかったとしても，必要なスピルコードはたかだか3個（生存区間中の定義・参照回数）だけである．

図6の各生存区間を並べ替えると，t1-b-t3-t4-a-t5-t7-t8-t2-y-x の順になる．

生存区間の割付け　取り出された生存区間に対して，並べ替えられた順番に，利用可能なレジスタを割り付

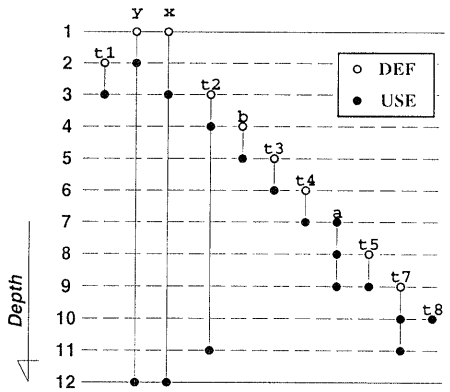


図6　生存区間解析の結果
Fig. 6　Result of lifetime analysis.

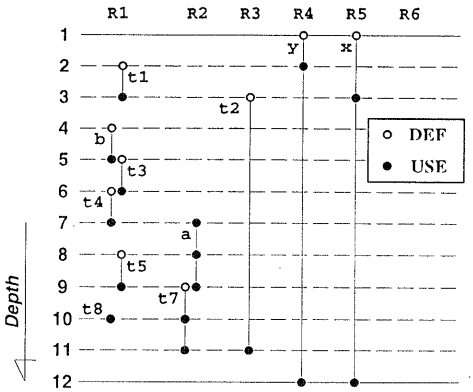


図7　レジスタが6個の場合
Fig. 7　Result of register allocation (available register = 6).

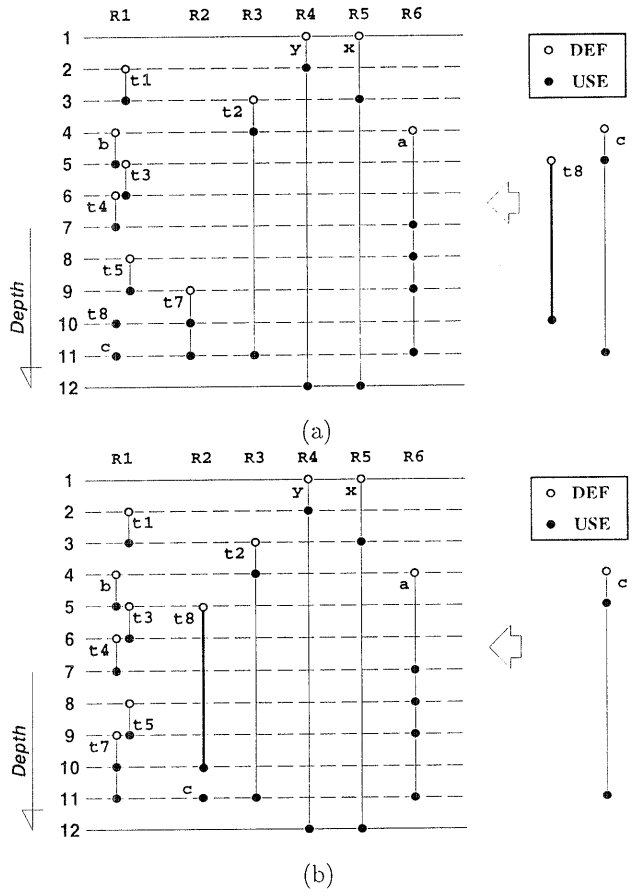


図8 レジスタ交換の例
Fig. 8 Example of swapping register.

けていく．図6の各生存区間を，レジスタ数が6個のプロセッサに割り付けると，**図7**のように割り付けられる．

割付け可能なレジスタが存在しないならば，次節に示す方法で，割付けが可能になるようにGPDGに手を加える．

4.3 スピル処理

生存区間に割り付けるレジスタがない場合，GPDGに対して何らかの変更を加えて割付けが可能となるようにしなければならない．その手段として，“レジスタの交換”，“ノードの移動”，“スピルコードの挿入”の三つを考える．これらの方法は，後者にいくほど，プログラムの性能を大きく落す可能性がある．したがって，これらの手段はこの順に行われ，前の処理ではレジスタを割付け可能にすることができない場合に，後者へと進んでいく．

生存区間の割付けは，自由度の低い方から順番に行われている．したがって，この時点で既にレジスタに割り付けられている変数は，これから割付けを行おうとしている変数よりも優先されなければならない．そのため，スピルコードの挿入など，プログラムの変更が必要になった場合にも，既に割付けの終了している変数は，できるだけ現在の状態を保持するようにし，これから割付けを行う生存区間に対して，スピルコードの挿入などを行うようにする．例えば，**図8**(a)において，これから割付けを行おうとしている生存区間**t8**は，左側の既に割付けの終了している生存区間に比べて，自由度が高く，割付けの優先順位が低い．もし，スピルコードの挿入が必要な場合には，この自由度の高い**t8**の生存区間に対してスピルコードを挿入するようにし，既に割付けの終了している，すなわち，より自由度の低い生存区間へのスピルコードの挿入は避ける．

レジスタの交換 レジスタの交換とは，既に割付けが終了している変数に対して，いずれかのレジスタの利用可能な区間が広がるように，変数の割り付けられて

いるレジスタを，他の変数と入れ換えることである．図8(a)の状態において，図横の変数 t8 をレジスタ R2 へ割り付けることはできないが，レジスタ R1 とレジスタ R2 の深さ9以降を入れ換え同図(b)のようにすることで，割付けが可能となる．

一般に，レジスタ X とレジスタ Y が，深さ d において，利用されていないか，または，割り付けられた生存区間の先頭であるとき，X と Y の深さ d 以降を交換することができる．図8(a)の例では，レジスタ R1 とレジスタ R2 を深さ9で入れ換えることは可能であるが，深さ10で入れ換えることはできない．

交換の具体的なアルゴリズムを図9に示す．

ノードの移動 図10(a)において，太線で表されたパス上の変数 i3 にレジスタを割り付ける場合を考える．図の左側にレジスタの利用状況と変数 i3 の生存区間を示しているが，変数 i3 が利用可能なレジスタは存在しないことがわかる．そこで，命令Aから命令Bへ，同図(b)のように仮の依存エッジを挿入し，Bを同図(b)の位置まで移動する．この操作によって，変数 i3 を変数 R2 へ割り付けることが可能となる．

具体的には，図9と同様のアルゴリズムにより，いずれかのレジスタの利用可能な区間を広げ，その空いている区間の直前の命令から，割り付けようとする生存区間の先頭の命令へ，また，割り付けようとする生存区間の最後の命令から，空いている区間の直後の命令へ依存エッジを張り，そのレジスタへ割り付ける．この操作によってクリティカルパスが伸びてしまった場合には，操作を元に戻し，次のスピルコードの挿入へ進む．

このノード移動は，他の割付けの終了していないノードの自由度をも圧迫する．そのため，過度のノード移動は，それ以降のレジスタ割付けに悪影響を及ぼすことが考えられる．そこで，クリティカルパスを延ばさないという条件に加え，ノード移動に次の条件を加える．

```
SwapRegister(生存区間 L)
begin
  X := L の先頭の深さで利用可能なレジスタ
  d := L の深さ以降で X が始めて利用不可能となる深さ
  while d < L の最後尾の深さ do
    Y := d 以降の利用が可能なレジスタ
    Y が見つからなければ終了
    X と Y の d 以降を入れ換え
    d := X が次に利用不可能となる深さ
  end
end
```

図9 レジスタ交換のアルゴリズム
Fig. 9 Algorithm of swapping register.

条件

ノード移動後の GPDG において，そのノード移動によって割付けが可能となったノードの集合を N_1，割付けが終了していないノードの集合を N_2 とするとき，

$$n \in N_1, m \in N_2, Free(n) > Free(m)$$

を満たす n，m が存在してはならない．

スピルコードの挿入 上記の“レジスタ交換”，“ノードの移動”のいずれの手法を用いても割付けが可能とならない場合には，プログラム中に適当なスピルコードを挿入する必要がある．

具体的には，割り付けようとしている生存区間において，変数の値を定義する命令の直後にスピルアウト命令を，その値を参照する命令の直前にスピルイン命令を挿入する．これにより，その生存区間は複数の小さな生存区間に分割され，割付けが可能となる．例えば，図8(b)において，変数 c を割り付ける場合，1個のスピルアウト命令と2個のスピルイン命令が挿入され，3個の生存区間に分割される．

それ以上分割できない場合（ロード命令とその値を参照する命令のみの生存区間など）には，先に述べたノードの移動と同じ方法で，命令間に仮の依存エッジを張り，クリティカルパスを延ばしてでも割付けを実行する．

4.4 割付け経路の再ソート

前述のように，本手法ではGPDGを自由度に応じていくつかの経路に分割し，自由度の低い経路から優先的にレジスタ割付けを行っていく．しかしながら，各ノードの自由度はレジスタ割付けを行っていくことによって，変化していく．そのため，ある経路を割り付けている最中に，その副作用によってその経路よりも自由度の低い経路が生じる可能性がある．

現状では，処理時間を考慮して，上のような可能性は無視している．すなわち，割付けを始めた経路は，そこに含まれるすべてのノードのレジスタ割付けが完了するまで割付けを続け，それが終了した時点で，残った経路の再ソートを行い，その中で最も自由度が低い経路を次の目標経路としている．

5. 関連研究との比較

文献8)~10)では，プログラムに対して命令間の依存関係を表すグラフを作成し，そのグラフの各ノード（各インストラクション）を先頭から順番にたどりながらレジスタを割り付けていく．このとき，レジス

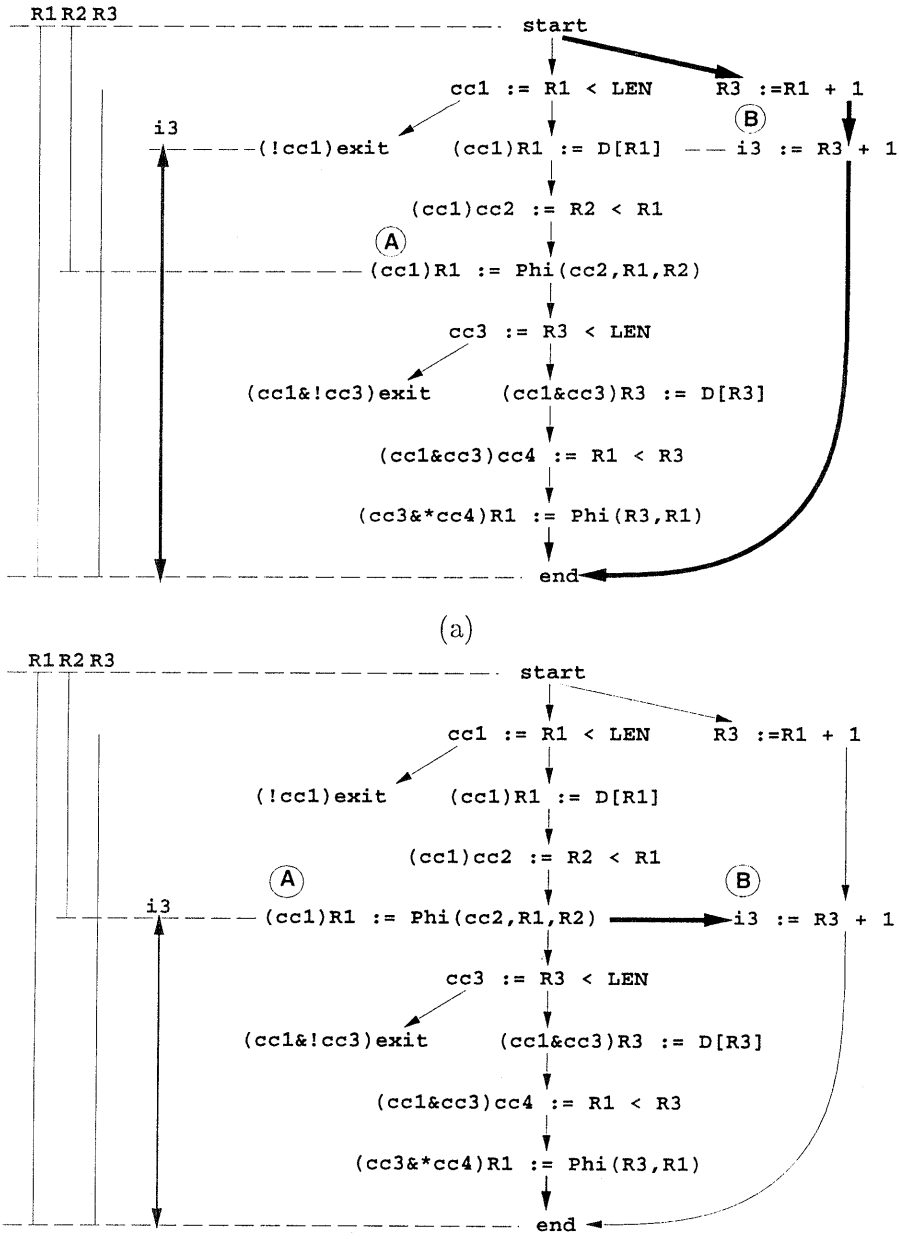


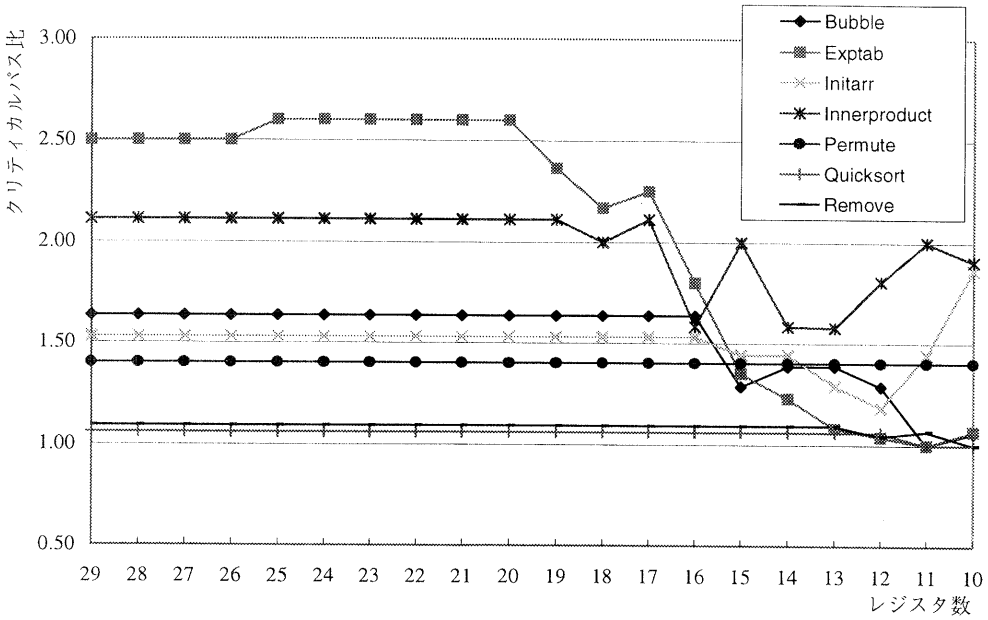
図 10 ノード移動の例
Fig. 10 Example of moving node.

タを割り付けることによって，それまで依存関係のなかった命令間に，逆依存が発生しないように留意しながら割付けを行っている．また，文献 8) においては，Prepass（コードスケジュールを先に行う），Postpass（レジスタ割付けを先に行う）の性能比較を行っており，Prepass の方がよい値を残すことが述べられている．

プログラムをグラフ構造で表し，干渉グラフに命令の並列実行を保証するためのエッジを付加する手法は，文献 12) にも述べられている．一般には，このようなエッジを加えることで，変数間の干渉度は高くなり，より多くのレジスタを必要とすることになってしまう．これにより，大量のスピルコードを発生させてしまっては，プログラムの実行時間を逆に延ばしてしまうことになりかねない．そこで，文献 12) の手法で

	使用可能なレジスタ数																			
	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	
Bubble	11	11	11	11	11	11	11	11	11	11	11	11	11	11	14	13	13	14	18	17	本手法
	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	従来手法+R.R.
	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	従来手法
Exptab	10	10	10	10	10	10	10	10	10	10	11	12	12	15	20	22	25	26	28	30	本手法
	25	25	25	25	26	26	26	26	26	26	26	26	27	27	27	27	27	27	28	30	従来手法+R.R.
	31	31	31	31	33	33	33	33	33	33	33	33	33	33	33	33	33	33	35	40	従来手法
Initarr	32	32	32	32	32	32	32	32	32	32	32	32	32	32	34	34	38	44	43	45	本手法
	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	52	62	84	従来手法+R.R.
	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	49	52	62	84	従来手法
Innerproduct	18	18	18	18	18	18	18	18	18	18	18	19	18	24	19	24	26	26	28	32	本手法
	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	41	47	56	61	従来手法+R.R.
	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	41	47	56	61	従来手法
Permute	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	本手法
	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	従来手法+R.R.
	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	従来手法
Quicksort	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	18	18	本手法
	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	従来手法+R.R.
	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	従来手法
Remove	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	46	45	48	本手法
	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	従来手法+R.R.
	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	53	従来手法

(a)



(b)

図11 クリティカルパスの比較
Fig. 11 Comparison of critical path length.

は，干渉グラフの彩色が不可能であった場合には，スピルコードの挿入だけでなく，命令の並列実行を諦めて干渉グラフの干渉エッジを取り除くことも考えている．これと同じことを，本手法ではノード移動で行っていることになる．しかし，グラフ上での割付け手順そのものは本手法とは大きく異なっている．

これらの方法に対して，本手法ではプログラムをGPDGで表現した場合の，各ノードの自由度をレジスタ割付けに利用している点が特徴である．ノードの自由度に着目することで，スピルコードが挿入された場合のクリティカルパスの変化を予測し，スピルコードが発生した場合にクリティカルパスを大きく伸ばし

	使用可能なレジスタ数															
	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	
Bubble	41	41	41	41	41	41	41	41	41	45	43	45	45	51	54	本手法
	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	従来手法
Innerproduct	86	86	86	86	86	86	88	88	92	92	94	97	103	110	120	本手法
	86	86	86	86	86	86	86	86	86	86	86	88	90	92	94	従来手法
Remove	83	83	83	83	83	83	83	83	83	83	83	83	87	91	99	本手法
	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	従来手法

図12 コードサイズの比較
Fig. 12 Comparison of code size.

てしまう可能性のあるノード（=自由度の低いノード）に優先的にレジスタを割り付ける．これにより，プログラムのクリティカルパスが大きく伸びてしまうことを抑制している．また，各変数からみた場合，その生存区間を一度に割り付けるのではなく，同一の変数であっても生存区間中で自由度の低い部分は優先的に，自由度の高い部分は後回しにされて割付けが行われる．他の手法では，各変数はその生存区間を一度に割り付けてしまおうとするため，自由度の高い部分での干渉によって，自由度の低い部分へスピルコードが挿入される可能性がある．

6. 評　価

本手法と，グラフ彩色法を用いた従来のレジスタ割付け手法との性能比較を行った．**図11**および**図12**にその結果を示す．プログラムはStanfordベンチマーク中のいくつかの関数から，その最内ループを取り出して利用した．図中の“従来手法”とは，現在，最も広く利用されている文献1)，2)に述べられた手法である．また，“従来手法 +R.R.”とは，上の手法が終了した後に，レジスタのリネームを行い，割付けによって発生した逆依存をできる限り取り除いたものである．図11 (a)は割付け後のGPDGのクリティカルパスを表し，図11 (b)は本手法を利用した場合のクリティカルパスの長さと，“従来手法 +R.R.”を利用した場合のそれとの比を表したグラフである．また，図12には，レジスタ割付け後の命令数を示した．なお，各命令はLOAD，STORE命令が2クロックで，その他の命令はすべて1クロックで実行可能とした．

レジスタ数が十分である場合にも，従来の手法では最適な結果が得られないことが，図11 (a)の表からわかる．このような場合，本手法を利用することで1.5倍程度速度が向上している．これは，前述の通り，割付けによって発生する逆依存が，プログラムの並列性を失わせてしまうためである．また，“従来手法 +R.R.”の結果が，本手法ほどではないことから，発生した逆依存を取り除くためにレジスタのリネームを行っても，完全に取り去ることはできないことがわかる．これらのことから，命令レベル並列プロセッサ用のコンパイラでは，レジスタ割付けにおいて発生する逆依存を考慮することが重要であると思われる．

レジスタ数が少ない場合には，速度の向上率が多少落ちてしまうが，それでも，割付け後のプログラムのクリティカルパスは確実に短くなっている．ところが，図12から，プログラムのコードサイズは大きくなっていること，すなわち，より多くのスピルコードが挿入されていることがわかる．コードサイズが増えているにも関わらず，クリティカルパスが短くなっているのは，スピルコードが，他の命令と並列に実行される位置にうまく挿入されていることを示している．

プログラムによって，速度の向上率に差異が見られるが，一般に，自由度の高い命令が多く存在するようなプログラムほど，よい結果が得られている．命令の自由度が全体に低いプログラムでは，スピルコードのコストが非常に大きくなってしまうため，プロセッサの並列性をうまく利用できていないものと思われる．

また，プロセッサのレジスタ数が減少しているにも関わらず，最終結果のクリティカルパスが短くなっている場合がある．本手法は最終結果の最適性を保証するものではなく，各生存区間はそれぞれの時点で局所的に最適な割付けが行われている．そのため，上述のようなことが起こり得るが，レジスタ数の変化に対して，クリティカルパスが極端に増減したりすることはなかったため，大きな問題とはならないと考えられる．

7. おわりに

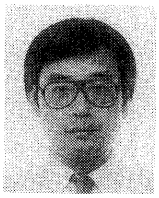
命令レベル並列プロセッサにおいては，レジスタ割付けフェーズにおける逆依存の発生によって，プログラムの並列性を落とさないようにすることが必要である．本稿では，そのための手法として，GPDGと呼ばれるデータ構造によってプログラムを表現し，その上でクリティカルパスへの影響の強い部分から優先的にレジスタ割付けを行っていく手法を提案し，従来手法との比較を行った．

評価結果から，並列性を持つプロセッサにおいては本手法が有効であることがわかる．逆に，従来の割付け手法をそのまま利用した場合には，プログラムの並列性が損なわれてしまい，プロセッサの性能を十分に引き出すことができていない．すなわち，命令レベル並列プロセッサにおいては，このような新しい手法の利用が必要になると言える．

参考文献

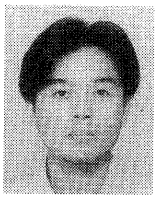
1) Chatin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopplins, M.E. and Markstein, P.W.: Register Allocation via Coloring, *Computer Languages 6*, pp.47–57 (1981).
2) Chaitin, G.J.: Register Allocation & Spilling via Graph Coloring, *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pp.98–105 (Jun. 1982).
3) Callahan, D. and Koblenz, B.: Register Allocation via Hierarchical Graph Coloring, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp.192–203 (Jun. 1991).
4) Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: An Efficient Method of Computing Static Single Assignment Form, *Conference Record of the Sixteenth ACM Symposium on the Principles of Programming Languages*, pp.25–35 (Jan. 1989).
5) Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The Program Dependence Graph and Its Use in Optimization, *ACM Trans. Prog. Lang. Syst.*, Vol.9, No.3, pp.319–349 (Jul. 1987).
6) 古関，小松，深澤：命令レベル並列アーキテクチャのための大域的コードスケジューリング技法とその評価，並列処理シンポジウム，JSPP '94 論文集，pp.1–8 (Aug. 1994).
7) Chang, P.P., Lavery, D.M., Mahlke, S.A., Chen, W.Y. and Hwu, W.W.: The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors, *IEEE Trans. Comput.*, Vol.44, No.3, pp.353–370 (1995).
8) Goodman, J.R. and Hsu, W.C.: Code Scheduling and Register Allocation in Large Basic Blocks, *International Conference on Supercomputing*, pp.442–452 (July 1988).
9) Bradlee, D.G., Eggers, S.J. and Henry, R.R.: Integrating Register Allocation and Instruction Scheduling for RISCs, *International Conference on Architectual Support for Programming Languages and Operating Systems*, pp.122–131 (Apr. 1991).
10) 立花，谷口，雨宮：データフロー解析による関数型言語 Valid のコンパイル法，情報処理学会論文誌，Vol.30, No.12, pp.1628–1638 (Dec. 1989).
11) Norris, C. and Pollock, L.L.: A Scheduler-Sensitive Global Register Allocation, *Proceedings of the ACM SIGPLAN '93 Conference on Supercomputing*, pp.804–813 (1993).
12) Pinter, S.S.: Register Allocation with Instruction Scheduling: A New Approach, *Proceedings of the ACM SIGPLAN '93 Conference on Programming Languages Design and Implementation*, pp.248–257 (1993).

(平成 7 年 2 月 27 日受付)
(平成 7 年 10 月 5 日採録)



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．



神力 哲夫

1969 年生．1993 年早稲田大学理工学部電気工学科卒業．1995 年同大大学院理工学研究科修士課程修了．同年大日本印刷（株）に入社．現在，同社画像研究所に勤務．建材分野における画像処理の研究に従事．

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．

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．