解　説

# VLIW 計算機のためのコンパイラ技術†

中谷　登志男††

## 1. はじめに

計算機の性能を向上させるアプローチとして重要になりつつある VLIW (Very Long Instruction Word) 計算機のコンパイラ技術について解説する．VLIW 計算機は最近まで，特に科学技術計算を対象として開発されてきたという背景がある．そのためここでは，それらを中心に話をすすめることにする．汎用を目的とした VLIW 計算機と，そのコンパイラ技術についてはこれからの研究課題である．

ここではじめに VLIW 計算機の特徴，思想，長所，及び他のアーキテクチャの問題点などを簡単に説明する．次にプログラムの中にある並列度について一般的に分かっていることを述べる．第三に，これまで開発された VLIW 計算機のためのコンパイラ技術のうち，主な三つの手法，トレース・スケジューリング法，パーコレーション・スケジューリング法，及びソフトウェア・パイプライニング技法について，手法の概略とその問題点を中心に解説する．

## 2. VLIW 計算機

### 2.1 特　徴

VLIW 計算機[1)~3)]の特徴は，1)毎サイクル一つの長命令語を取り出す制御機構をもつ（命令カウンタは一つということ），2)一つの命令は複数のオペレーションからなる，3)各オペレーションはコンパイル時に予測可能な（少ない）サイクル数で実行可能である，などである．各オペレーションは単純な RISC (Reduced Instruction Set Computer) 命令に相当し，レジスタ間のオペレーションと単純なアドレシング・モードのロード（メモリからレジスタへの移動）及びストア（レジスタからメモリへの移動）・オペレーションからなる．言い換えると，VLIW 計算機は複数の RISC オペレーションを毎サイクル実行できる単一プロセッサである．

ここで重要なことは，長命令語の中に含まれる複数のオペレーションが並列実行可能であることをコンパイラが常に保証しているという点である．言い換えると，一つの長命令語の中で1)オペレーション間で演算器及びデータ経路の共有がない，2)どのオペレーションも他のオペレーションの結果を必要としない，という二つの制限が常に守られていなければならない．

### 2.2 RISC との共通思想

ところで，RISC はコンパイラの観点から，垂直型マイクロプログラム・アーキテクチャを見直したものであるが，VLIW 計算機もコンパイラの観点から，水平型マイクロプログラム・アーキテクチャを見直したものである．重要な点は，どちらもコンパイラがプロセッサ内の各ハードウェア・リソースを直接管理できるアーキテクチャを採用することにより，プロセッサのハードウェア利用効率をあげ，全体として処理能力を向上させている点である．

特に，従来の水平型マイクロプログラム・アーキテクチャでは，データ・パスや命令フィールドの一部を共有しているために，複数の演算器を同時に独立して利用することができない場合が少なくなかった．VLIW 計算機では，命令長やレジスタのポート数を十分大きくとることにより，これを可能にしている．また，従来の水平型マイクロプログラム・アーキテクチャでは，命令長はせいぜい 100 ビット前後であったが，VLIW 計算機では 1000 ビットぐらいのものまで考えられている[1),2)]．これは，コンパイラで取り出せる並列実行可能なオペレーションの数に大きな違いがあるということでもある．

さらに，VLIW 計算機には RISC との共通点が多い．1)プログラムは，C や Fortran などの高級言語で記述することを前提としている点や（直接，VLIW 計算機の機械語（長命令語）でプログラムを書くことはほとんど不可能に近い），2)どちらも，プログラム

† Compilation Techniques for VLIW Machines by Toshio NAKATANI (IBM Tokyo Research Laboratory).

†† 日本アイ・ビー・エム(株)東京基礎研究所

とそれを実行するハードウェアとのあいだにマイクロプログラムなどの中間層が介在しない点，などである．このため，コンパイラはプログラムをRISCオペレーションに分解し，各RISCオペレーションを計算機のハードウェア・リソースに割り当て，管理することが可能なのである．

### 2.3 他のアーキテクチャの問題点

並列計算機では，通信や同期による遅延を最小限に抑えると同時に，各プロセッサを最大限に有効利用しなければならない．そのために，コンパイラは比較的独立して実行できる部分をプログラムから抽出しなければならない．実用段階では，コンパイラが取り出せる部分は，データ依存のない(最も内側の)単純なループに限られているのが現状である．

ベクトル計算機では，単一のオペレーション（ベクトル操作）で多くのデータを続けて処理する（ベクトル化できる）部分が豊富になければ，効率の良い実行ができない．このため，コンパイラはプログラムの中から規則性のあるデータ構造や制御構造を探す必要がある．昨今こういったベクトル化コンパイラ技術の進歩は著しいものの，プログラムの中でコンパイラが最適化できる部分に限りがある．

### 2.4 長 所

これに対して，VLIW 計算機では通信や同期による遅延がなく，複数の異なるオペレーションを毎サイクル，それぞれ個々のデータに対して独立に操作できるという利点がある．これにより，コンパイラは並列計算機やベクトル計算機よりもプログラムのより広範囲な部分を最適化の対象とすることができる．

## 3. 複数の命令を取りだすということ

単一プロセッサでは，毎サイクル，新しい命令を一つデコードするのであるが，ここでは仮に，複数の命令を同時にデコードできるようにした場合（実際，このようなプロセッサは，スーパスカラ計算機[3]と呼ばれている）を考えてみよう．基本的な問題点は，取りだした複数の命令が並列実行可能であるかどうか，ということである．

さまざまなプログラムを解析した結果，平均で高々二つの命令が並列実行可能であるという事実が，1970年代の初頭より明らかになっている[4)~6),8)]．そして，並列実行可能な命令とは，ループでのインデックスの更新と本来実行したい演算そのものの二つの組合せ以外にはないことが多い．RISC では，コンパイラが命令の順序を入れ換えることにより，メモリからレジスタへのロード命令や分岐命令などの遅延サイクルを利用して，インデックス命令が実行される．

ところで，なぜ実際のプログラムの中に，並列度がほとんどないのであろうか？その理由は，分岐命令があると，それ以前の命令とそれ以後の命令とを並列実行できないからである．分岐命令の頻度は平均，数命令に一回ぐらいと多いので，結局，二つの分岐命令の間に並列実行できる命令は，高々二つということになる．それでは分岐命令は減らせないのであろうか？残念ながら，一般的には難しい．

しかし，科学技術計算などのプログラムの中には，かなりの分岐命令でどちらに分岐するかが予測できるという場合がある．もし各分岐命令で分岐先を予測して分岐命令を実質的になくしてしまえれば，全て確実に実行される長い命令列（トレース (Trace) と呼ばれている）が取りだせることになる．そしてこの一連の長い命令列の中には，並列実行可能な命令がかなり多く含まれている．実際，各分岐命令でプログラムがどちらに分岐するかが前もって分かっていたとすれば，50 倍を超えるスピード・アップが得られる可能性がある（ただし，一つぐらいの分岐命令をスキップしても，せいぜい2～3倍ぐらいのスピード・アップ）という報告もある[6),7)]．ところが，1970 年代には，実用的ではないという理由でそれ以上の試みはなされなかった．

## 4. トレース・スケジューリング法

1980 年代に入って，分岐先を予測することにより得られる並列度を利用して，プログラムの実行時間の飛躍的な短縮を図る，トレース・スケジューリング (Trace Scheduling) 法[9),10)]と呼ばれる試みが始まる．主に科学技術計算のプログラムを対象とすることにより，分岐予測がおおむね正しい場合に限っている．トレース・スケジューリング法は，1) トレースの選択，2) トレースの並列化及びコード生成，3) ブックキーピング (Bookkeeping) (プログラムの他の部分との意味合わせ)，の三つのステップからなる．この三つのステップを，最も実行される可能性の高いトレースから順々に，各トレースに適用する．

### 4.1 トレースの選択

トレースを選択するには，まず分岐確率を得なければならない．これには，三つの方法がある．第一は，プログラマがソース・プログラム中に，各分岐点ごと

に確率を明示してやる．第二は，自動プロファイラ（プログラムの実際の実行カウント）を利用する．第三は，コンパイラが，各分岐にデフォルトとして 50-50 の確率を与える．そして，各オペレーションの予想実行回数を，今求めた確率と各ループの予想繰返し数とを掛け合わせることにより得る．

次に，最も予想実行回数の多いオペレーションを選び，それをトレースの核とする．核のオペレーションに続くオペレーションの中で，最も予想実行回数の多いオペレーションをトレースに加える．ただし，すでにトレースに加えられているオペレーションや（ループでの）戻り分岐に続くオペレーションは対象から外す．核のオペレーションの前のオペレーションについても同様に繰り返す．このようにして一つのトレースができあがる．

要約すると，トレースの選択は，最も内側のループから始まり，まずそのループの中だけでトレースを順に選ぶ．そして，最も内側のループが全て終わったら外側のループに移る．ループの内側と外側ではそれぞれ別々のトレースとなる．図-1 を参考にしていただければ，全体のトレースの様子がお分かりいただけると思う．

### 4.2 トレースの並列化及びコード生成

一つのトレースが選ばれると，そのトレースを並列化し実際の VLIW 計算機のコードを生成する．データ依存と次に述べる先行制約（Precedence Constraint）の順序関係でトポロジカル・ソートを行い，各オペレーションを順に実際のハードウェア・リソースに割り当てていく（こういう手法を一般的にリスト・スケジューリング (List Scheduling) と呼ぶこともある）．

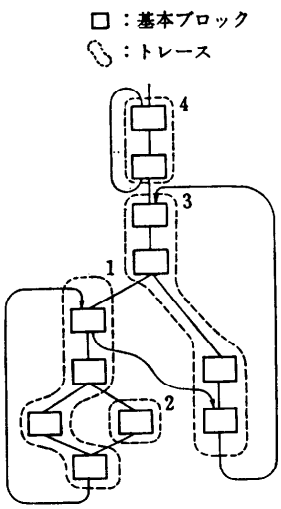


図-1 トレースの選択

データ依存については，特にここで説明する必要はないと思うが，先行制約について二つの例をあげておく．まず第一は，図-2 のように，予測した分岐先のオペレーションがその分岐命令の前に実行され，さらに予測と反する分岐先で参照される変数を破壊する場合である．そういう場合を避けるために，そのオペレーションと分岐命令との間に先行制約の順序関係をつける．

第二は，図-3 のようにトレース中に，ある変数を参照するオペレーションとそれを破壊するオペレーションが共存する場合である．このような場合にも，二つのオペレーションの間に先行制約の順序関係をつける．

### 4.3 ブックキーピング

以上のように並列化されたコードと，プログラムのその他の部分との意味的正しさを保証するために，次の二つのコピー操作を行う必要がある．第一に，分岐命令の前に実行していなければならないオペレーションが，分岐が予測と反したために，実行されない場合である．この場合には，その実行していなければならないオペレーションを，分岐先の次のトレースの始め

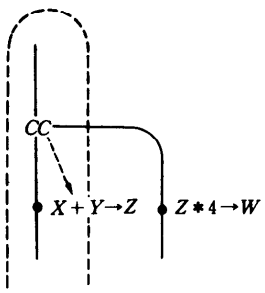


図-2 先行制約（その 1）

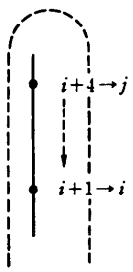


図-3 先行制約（その 2）

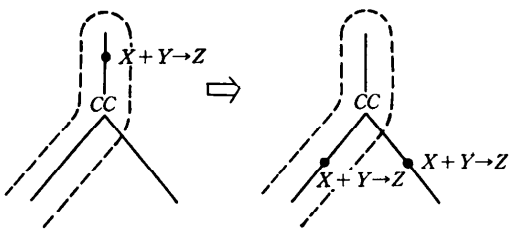


図-4 ブックキーピング（その1）

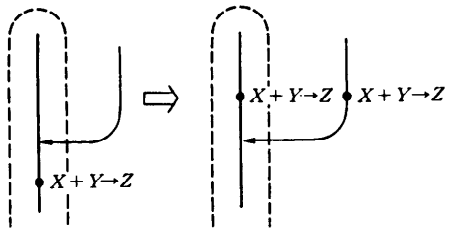


図-5 ブックキーピング（その2）

に，コピーして挿入しなければならない(図-4 参照)．第二に，別のトレースから分岐して現在のトレースに合流してくる場合に，すでに実行されていなければいけないオペレーションがまだ実行されていない場合である．この場合にも，未完了のオペレーションを合流してくるトレースの合流点に，コピーして挿入しなければならない（図-5 参照)．

### 4.4 ループの扱い

以上みてきたように，トレース・スケジューリング法では，ループはそのままの形で保存し，ループの中だけをいくつかのトレースに分割する．これだけでは実際，長いトレースが取りだせないため，トレース・スケジューリング法では，トレースの選択に先立って，ループを何回か展開する（ループ展開 (Loop Unrolling) と呼ばれる)．ループ展開により，プログラムのコード・サイズを増大させることになるが，並列実行可能なオペレーションを増やし，全体としてプログラムの実行時間を短縮することができる．

### 4.5 トレース・スケジューリング法の問題点

まず第一に，プログラムのコード・サイズが指数的に大きくなる可能性があるという問題である[10]．これは，ブックキーピング時に挿入されるオペレーションのコピーが問題なのである．特にループ展開によるコード・サイズの増大とあいまってトレース・スケジューリングの大きな問題となっている．

第二に，分岐予測の問題である．科学技術計算などでは分岐が予測できるものが多いとしても，その他のプログラムではどちらに分岐するか分からない場合が多い．特に，前方に分岐する場合は予測がむつかしく，予測がはずれた場合の効率の低下が問題となる(図-6 参照)．

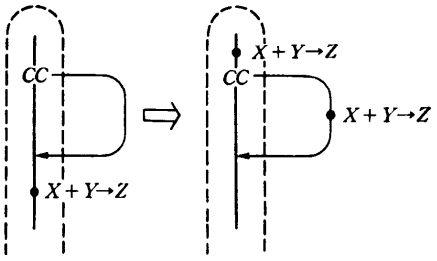


図-6 効率の悪いコピー

第三に，ループの取扱いである．十分な回数のループ展開を行わないと，満足のゆくスピード・アップが得られないが，反面コード・サイズの増大が問題となる．また，ループ展開だけではループの繰返しによる負荷の問題の解決にはならない．たとえば，パイプラインの初期及び終了時の効率低下がループ境界ごとに起こる．その解決には，後半に述べるソフトウェア的なループの完全なパイプライン化（ソフトウェア・パイプライニング）が必要である．

### 4.6 トレース・スケジューリング法の改善

トレース・スケジューリング法の発表以来，いくつかの改善案が考えだされた．第一の方法は，コード・サイズの増大を抑えるために，プログラムをまず木（ツリー）に分割し，各木にトレース・スケジューリングを適用するツリー・コンパクション (Tree Compaction) 法である[11]．この方法では，木のルートを超えてオペレーションを上げることができないので，ブックキーピング時のコピーが少なくなるが，反面，得られる並列度も限られるという問題がある．

第二の方法は，図-6 の例のようなトレース・スケジューリング法における効率の悪いコピーの生成を防ぐために，トレースの一つの基本ブロック（実行されるときは，この中の全てのオペレーションが実行される連続した部分）のコードを生成するたびに，それ以降のトレースを選び直し，選んだトレースの最初の基本ブロックに入れるオペレーションの候補をトレースの中からだけではなく，今選んだ基本ブロックをルートとする DAG ((Directed Acyclic Graph) から選び，これを繰り返していく SRDAG (Single Rooted DAG) 法である[12]．

この方法の利点は，トレース・スケジューリング法ではできなかった，複数のパスに存在する共通のオペ

レーションは可能なかぎり一つにまとめられること(ユニファイ (Unify) と呼ぶ) が可能になった点である．しかし反面，ブックキーピング時におけるプログラムの意味合わせが非常に複雑になるという欠点がある．これは，トレース・スケジューリング法では一つのパスのみを考慮に入れればよかったのに対し，SRDAG 法では，複数のパスを同時に考慮に入れなければならないからである．

## 5. パーコレーション・スケジューリング法

トレース・スケジューリング法での経験を生かして，まったく新しい視点から，1985年に考えだされたのがパーコレーション・スケジューリング法である．パーコレーション・スケジューリング (Percolation Scheduling) 法[13]は，4つの単純な基本操作によりプログラムの意味的正しさを保ちながらプログラムを少しずつ変換して並列化を図ろうとする試みである．以下に述べる基本操作をどう利用するかはまったく自由であるが，プログラムの終点から始点に向かって，コードを上げていく(パーコレート (Percolate) する)ことを目的としている．

### 5.1 計算モデル

パーコレーション・スケジューリング法で基本となる計算モデルは，プログラムをノードとノード間を結ぶエッジからなる有向グラフに表したもので (ループのあるプログラムは扱えない)，分岐エッジは，分岐関係に対応する．グラフ中の各ノードは長命令語に対応する．次に述べる基本操作では，各エッジにそって，一つ一つのオペレーションあるいは分岐を前方向に移動する操作が中心となる．

パーコレーション・スケジューリング法では，トレース・スケジューリング法と比較して，分岐の扱いに関してより一般化されている．図-7 のように，トレース・スケジューリング法では，トレースを選ぶことにより深さ方向に分岐を進めることしかできないのに対して，パーコレーション・スケジューリング法では，水平方向にも同時に分岐を進めることが可能である．

### 5.2 基本操作

基本操作 1：オペレーションの移動

二つの連続するノード間で，後のノードのオペレーションが前のノードのどのオペレーションにもデータ依存しない場合，そのオペレーションを前のノードに移動できる．このとき，問題になるのは後のノードに

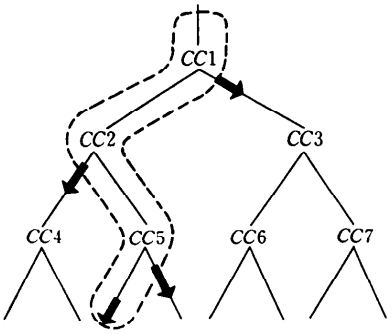


図-7-1 分岐木 (トレース・スケジューリング法)

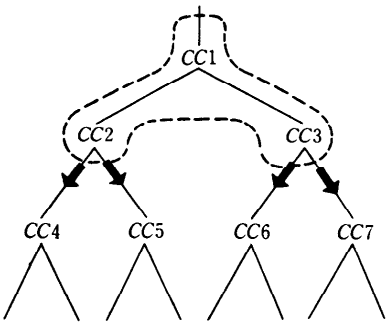


図-7-2 分岐木 (パーコレーション・スケジューリング法)

合流してくるエッジがある場合で，トレース・スケジューリングでみたように，プログラムの意味的正しさを保ってやらなければならない．パーコレーション・スケジューリング法では，図-8 のように，後のノードのコピーを作ってから，そのオペレーションを移動する．

基本操作 2：分岐の移動

二つの連続するノード間で，後のノードの分岐が前のノードのどのオペレーションにもデータ依存しない場合，その分岐を前のノードに移動できる．このとき，オペレーションの移動と同様，後のノードに合流してくるエッジがある場合，図-9 のように，後のノードのコピーを作ってから，その分岐を移動する．また，分岐の移動の場合は，後のノードの残り部分を，図-9 のように，二つのノードに分ける必要がある．

基本操作 3：ユニファイ

この操作は，前記の二つの操作とは異なり，一つのノードとその分岐先の複数のノードとの間で同じオペレーションのコピーが複数あるとき，まとめて一つにユニファイして前のノードに移動する操作である．これは，前記の二つの操作と同様，後のノードに合流してくるエッジがある場合，図-10 のように，後のノー

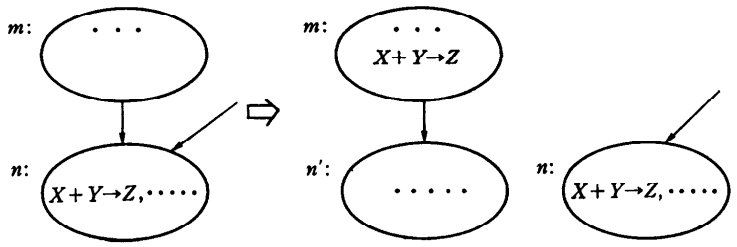


図-8 オペレーションの移動

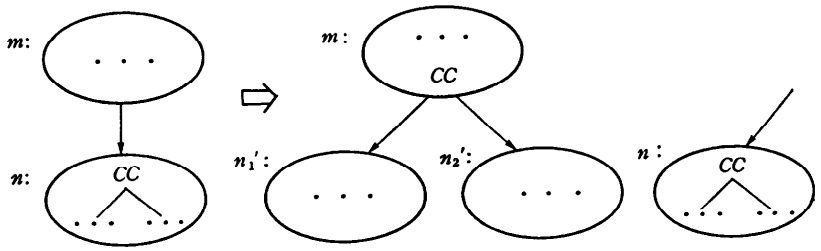


図-9 分岐の移動

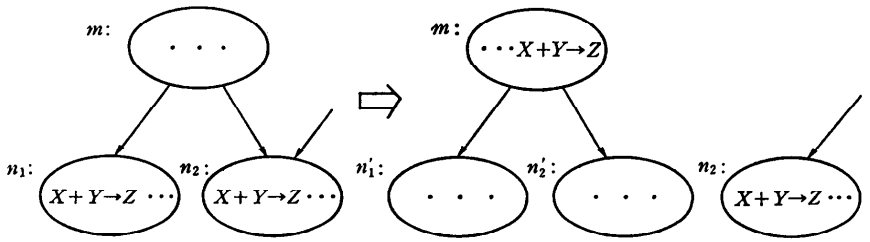


図-10 ユニファイ

ドのコピーを作ってからそのユニファイを行う.

基本操作 4: 命令の削除

この操作は, 前記の三つの操作とは異なり一つのノードごとに, 空になったかどうか調べて, 図-11 のように空のノードを削除する操作である.

### 5.3 パーコレーション・スケジューリング法の問題点

第一に, パーコレーション・スケジューリング法においても, プログラムのコード・サイズをいかに制御するかが問題となる. オペレーション及び分岐の移動にともなうノードのコピーが実用上問題となることが考えられる.

第二に, パーコレーション・スケジューリング法でもループの取扱いが問題である. トレース・スケジューリング法と同様, ループ展開だけでは, コード・サイズの問題及びループの繰返しによる負荷の問題の解決にならない.

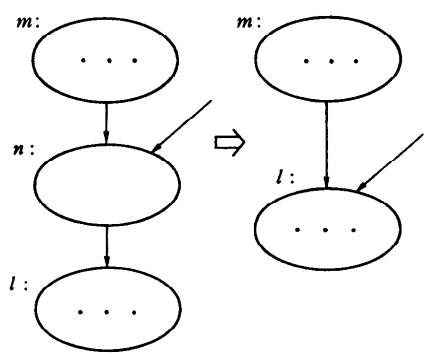


図-11 命令の削除

## 6. ソフトウェア・パイプライニング技法

これまでみてきた並列化法における共通の問題点は，ループの取扱いである．よりよいスピード・アップをはかるためには，ループの中を何回か展開し，ループの基本構造を変えずにループの中だけを並列化する，という方法であった．先にも述べた二つの問題，1)プログラムのコード・サイズの問題，2)ループの繰返しによる負荷の問題，をソフトウェア的に解決しようとする試み，ソフトウェア・パイプライニング(Software Pipelining)技法，が1980年代の初めより始まっている．

もともとマイクロ・プログラマの間では，手でマイクロ・プログラムの一部をパイプライン化することは行われていたのだが，本格的に自動化されはじめたのは最近になってからである．ただし初期のものは，条件分岐を含まない基本ブロックのみを対象としたソフトウェア・パイプライニング技法であったために実用的ではなかった．初期の方法は，次に述べるモジュロ・スケジューリング法に基づくものが多い．また，その次に紹介するハイアラキカル・リダクション法及びパーフェクト・パイプライニング法は，任意のループを扱えるようになったという点で一歩実用に近づいたといえる．

### 6.1 モジュロ・スケジューリング法

#### 6.1.1 概　略

仮に今，ループが$m$個のオペレーションからなっているとしよう．もしそのオペレーション（各オペレーションは一サイクルで実行できると仮定）を一つずつ実行したとすると，$m$サイクルかかる．すなわち，$m$サイクルに一回，ループが繰り返されることになる．これを，$t$ ($<m$) サイクルごとに，ループの新しい繰返しを始められないか？$t=1$ から順に試みて最小の$t$を見つけよう，という考えから生まれた方法をモジュロ・スケジューリング (Modulo Scheduling) 法と呼ぶ[14)~16)]．ループの中の毎 $t$ 番目のオペレーションが並列にスケジュールされるので，このように呼ばれている．

実際，前のループの結果を次のループの最初に必要とする場合や，対象とする計算機のハードウェア・リソースの制限から，毎サイクル，ループの新しい繰返しを始めることはできない．しかし，より小さな$t$で新しい繰返しが始められるほど，効率が良い．図-12の例は，$t=1$ でループの新しい繰返しが可能な例である．この方法は，先にも述べたように，条件分岐を含まない基本ブロックしか扱えなかったのであまり実用的ではなかった．

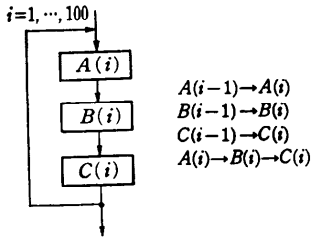


図-12-1 もとのループと各ステートメントの依存関係

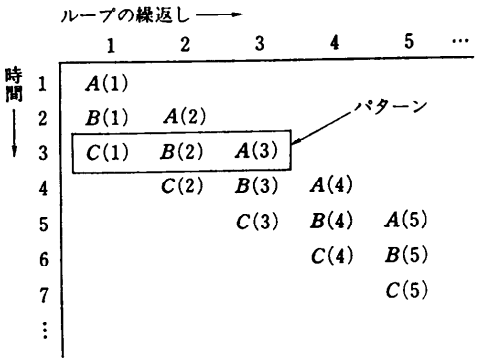


図-12-2 ループの最も効率の良い実行

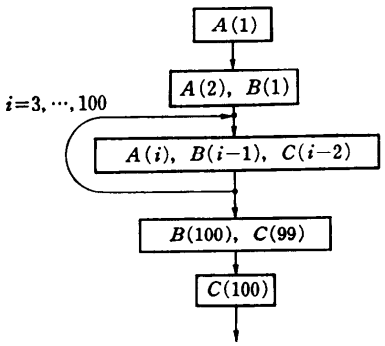


図-12-3 ソフトウェア・パイプライン化されたループ

#### 6.1.2 トレース・スケジューリング法への適用

トレース・スケジューリング法をつかって何回か展開したループの中に，繰り返されるパターンがないかを見つけ，それをパイプラインの安定状態として，ループにすればよいという考えは，トレース・スケジューリング法が考え出された当初からあった．しかし，これによって見つかる繰返しのパターンは非常に大きくなるという問題があり実用的ではない．

ループを一回だけ展開し，トレース・スケジュー

リング法を適応し，その後またループを巻き戻す，URCR (Unroll, Compaction, and Reroll) 法[17]と，その一般化である URPR (Unroll, Pipelining, and Reroll) 法[18]や GURPR 法[19]という手法も提案された．しかしこの方法では，コード・サイズの節約にはなるが，取り出せる並列度に限りがあるという問題がある．また，ループの中で，繰返しのたびにいつも同じパスを通るという仮定にも問題がある．

### 6.2 ハイアラキカル・リダクション法

#### 6.2.1 背 景

1987年になって，モジュロ・スケジューリング法を改良し，さらに条件分岐も含む任意のループを扱える，ハイアラキカル・リダクション (Hierachical Reduction) 法[20]が考案された．この方法は，シストリック・アレイのセルのために，効率の良いコードを生成する目的で開発された．

#### 6.2.2 概 略

第一に，先ほどの $t$ の下限と上限を求めてその間でモジュロ・スケジューリング法を行う．データの循環依存（サイクルを含むということ）をもつループを扱えるようにするために，データ依存グラフの強結合部分 (Strongly-Connected Component) を一つのノードとして順にスケジュールしていく．

第二に，ループの繰返しのたびに同じ変数を使っていると逆依存 (Anti-Dependence)（前のデータの参照が終わるまでそのデータを破壊できないなど）のために，先ほどの $t$ の値が大きくなってしまうという問題が起こる．これを防ぐために，ループの新しい繰返しのたびに新しいレジスタやメモリ・セルを割り当てる．これは，一種のスカラ・エクスパンション (Scalar Expansion) 手法で，モジュロ・バリアブル・エクスパンション (Modulo Variable Expansion) 法[20]と呼ばれている．

第三に，分岐を含むループを扱えるようにするために，IF-THEN-ELSE の分岐の IF 部分と ELSE 部分をそれぞれ独立して並列化し，その後 (NOP などを入れて) 同じ長さのコードにし，それをひとまとめにして一つの IF-THEN-ELSE のノードとして取り扱う．これをハイアラキカル・リダクション法と呼んでいる．ここで，リソースやデータ依存の制約は IF 部分と ELSE 部分のそれぞれの和として取り扱う．

#### 6.2.3 問 題 点

第一の問題は，IF-THEN-ELSE を一つのノードとして固定してしまうので，コードの移動性が限られてしまう．IF 部分と ELSE 部分がまったく異なる大きさ，リソース依存，データ依存をもつときには，効率の悪い並列化になってしまう．

第二の問題は，IF-THEN-ELSE と同時にスケジュールすることができるオペレーションが他にある場合は，IF 部分及び ELSE 部分にそのオペレーションをコピーしなければならない．これによるコード・サイズの増加が問題となる．

また，シストリック・アレイのセルという制限された小さなハードウェア・リソース（実験された対象は，二つのフローティング・オペレーション，一つのメモリ・オペレーション，一つの分岐，及び，入出力オペレーションを毎サイクル行えるプロセッサ）を対象としては成功したが，大きなハードウェア・リソースを対象としては，これらの点が大きな問題となりうる．

### 6.3 パーフェクト・パイプライニング法

#### 6.3.1 背 景

パーフェクト・パイプライニング (Perfect Pipelining) 法[21]は，パーコレーション・スケジューリング法によるループの取扱いを改良するために考案された手法である．

#### 6.3.2 概 略

まずループを何回か展開し，それからパーコレーション・スケジューリング法を適用して，プログラムの始点方向に（ループの同じ繰返しに属するオペレーションはそのままの順序で）各オペレーションを，全ての可能性のあるパスを通して，上げられるだけ上げていく．次に，プログラムの始点からはじめて，プログラムの各ノードと同じ機能のノードが他にないかを探す．同じノードが見つかったら一つにまとめる．これを繰り返し，最後に，同じ機能のノードが見つかっていないノードがまだあれば，ループを展開する回数を増やして，同じプロセスを繰り返す．

#### 6.3.3 問 題 点

このアルゴリズムが終了するためには，ループの同じ繰返しに属するオペレーションは固定長内に抑えておかなければならない．そのために，ループのそれぞれの繰返し内のオペレーションは全てもとの順序のままスケジュールするようにしている．ループのある一つの繰返し部分から開始し，それを途中で中断し，再開するということは許されない．条件分岐の IF 部分と ELSE 部分で，異なったスケジュールができないという効率の悪さが問題である．また，実用上アルゴリズムが収束する速さに問題があるように思える．

## 7. おわりに

以上，トレース・スケジューリング法，パーコレーション・スケジューリング法，そしてソフトウェア・パイプライニング技法を中心に，VLIW 計算機のコンパイラ技術をみてきた．トレース・スケジューリング法はすでに商用化されているが，後者の二つはまだ研究段階にある．特に，ソフトウェア・パイプライニング技法について今後の研究が期待される．

さて，これまでみてきたものはどれも，VLIW 計算機の適用範囲として科学技術計算のプログラムを対象に研究開発されたきた．しかし，一般の汎用プログラムを対象とした VLIW 計算機の研究開発はこれからである．予測の難しい条件分岐を多く含むプログラムを対象とした研究もその一つである．また，一つの適用業務のみを対象に最適化された VLIW 計算機も今後の面白い課題である．

RISC や VLIW 計算機など，これからの計算機では，コンパイラの技術による性能の向上がますます重要になっていく．いかにして高度なコンパイラ技術を蓄積していくかが今後の発展の鍵となるであろう．VLIW 計算機のためのコンパイラ技術はまさにその代表的な例である．

## 参考文献


VLIW 計算機については，

1) Fisher, J. A. [1983]. "Very Long Instruction Word Architectures and the ELI-512, In Proceedings of the 10 th Annual Symposium on Computer Architectures, ACM, pp. 140-150.
2) Fisher, J. A. [1984]. "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", IEEE Computer 17 (7), pp. 45-53.
3) Colwel, R. P., Nix R. P., O'Donnel, J. J., Papworth, D. B. and Rodman P. K. [1988]. "A VLIW Architecture for a Trace Scheduling Compiler", IEEE Transactions on Computers, C-37 (8), pp. 967-979.

プログラム中の並列度については，

4) Tjaden, G. S. and Flynn, M. J. [1970]. "Detection and Parallel Execution of Independent Instructions", IEEE Transactions on Computers, C-19 (10), pp. 889-895.
5) Foster, C. G. and Riseman, E. M. [1972]. "Percolation of Code to Enhance Parallel Dispatching and Execution", IEEE Transactions on Computers, C-21 (12), pp. 1411-1415.
6) Riseman, E. M. and Foster, C. G. [1972]. "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Transactions on Computers, C-21 (12), pp. 1405-1411.
7) Nicolau, A. and Fisher, J. A. [1984]. "A Measuring the Parallelism Available for Very Long Instruction Word Architectures", IEEE Transactions on Computers, C-33 (11), pp. 968-976.
8) Jouppi, N. P. [1989]. "The Nonuniform Distribution of Instruction-Level and Machine Paralelism and Its Effect on Performance", IEEE Transactions on Computers, C-38 (12), pp. 1645-1658.

トレース・スケジューリング法については，

9) Fisher, J. A. [1981]. "Trace Scheduling: A Technique for Global Microcode Compaction", IEEE Transactions on Computers C-30 (7), pp. 478-490.
10) Ellis, J. R. [1986]. "Bulldog: A Compiler for VLIW Architectures", The MIT Press.

ツリー・コンパクション法については，

11) Lah, J. and Atkins, D. [1982]. "Tree Compaction of Microprogramms", In Proceedings of the 16 th Annual Workshop on Microprogramming, ACM, pp. 23-33.

SRDAG コンパクション法については，

12) Linn, J. L. [1983]. "SRDAG Compaction—A Generalization of Trace Scheduling to Increase the Use of Global Context Information", In Proceedings of Annual Microprogramming Workshop", ACM, pp. 11-22.

パーコレーション・スケジューリング法については，

13) Aiken, A. and Nicolau, A. [1988]. "A Development Environment for Horizontal Microcode", IEEE Transaction on Software Engineering, 14 (5), pp. 584-594.

初期のソフトウェア・パイプライニング技法については，

14) Charlesworth, A. E. [1981]. "An Approach to Scientific Array Processing: The Architectural Design of the AP-120 B/FPS-164 Family", IEEE Computer, 14 (9), pp. 18-27.
15) Touzeau, R. F. [1984]. "A Fortran Compiler for the FPS Scientific Computer", In Proceedings of SIGPLAN Symposium on Compiler Construction, ACM, pp. 48-57.
16) Rau, B. R. and Glaeser, C. D. [1981]. "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing", In Proceedings of the 14 th Annual Workshop on Microprogramming, ACM, pp. 183-198.

トレース・スケジューリング法への適用については，

17) Su, B., Ding, S. and Jin, L. [1984]. "An

Improvement of Trace Scheduling for Global Microcode Compaction", In Proceedings of the 17th Annual Microprogramming Workshop, ACM, pp. 78-85.

18) Su, B., Ding, S. and Xia, L. [1986]. "An Extension of URCR for Software Pipelining", In Proceedings of the 19th Annual Microprogramming Workshop, ACM, pp. 94-103.

19) Su, B., Ding, S. and Xia, L. [1987]. "GURPR —A Method for Global Software Pipelining", In Proceedings of the 20th Annual Microprogramming Workshop, ACM, pp. 88-96.

ハイアラキカル・リダクション法については,

20) Lam, M. [1988]. Software Pipelining: An Effective Scheduling Technique for VLIW Machines", In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, ACM, pp. 318-328.

パーフェクト・パイプライニング法については,

21) Aiken, A. and Nicolau, A. [1988]. "Perfect Pipelining: A New Loop Parallelization Technique", In Proceedings of the 1988 European Symposium on Programing Technique, pp. 221-235.

(平成2年2月 23 日受付)