計算機アーキテクチャ　152－4
（２００３．３．１０）

VLIW計算機における効率の良い多重分岐の命令スケジューリング

陳 煜　　鈴木 貢　　渡邊 坦

電気通信大学 情報工学科

本研究では、プレディケーション機能と並列分岐処理機能を持つVLIW計算機を対象として、多重分岐の各条件間の依存関係の有無にあまり関わらず、多重分岐を並列的に実行できるように変換する命令スケジューリング方式を提案し、実装した。これにより、クリティカルパスを短縮でき、多数の並列的に実行できる分岐ブロックが形成でき、命令レベル並列性が向上する。各条件の依存関係をなくするため、予測確率が高い分岐命令を優先的にスケジューリングすることが可能となる。

コンパイラの最適化のレベルにより、効果が異なるが、実装した対象であるIA-64のコードに適用したところ、6-23%の性能向上を確認できた。

さらに、case数が多いswitch文に対しては、提案した方式を拡張するn分探索法を考案し、実験したところ、4%の性能向上が得られた。

Scheduling of Multiple Branches for VLIW Processors

Chen Yu, Suzuki Mitsugu and Watanabe Tan

Computer Science, University of Electro-Communications

For VLIW processors supporting predication and multiway branches,we present a method to schedule instructions of multiple branches without regarding dependences among the conditions.It has been implemented on a VLIW processor IA-64.

By the method,we can reduce the length of critical path,and extract multiple branch blocks that can be executed in parallel,therefore ILP is improved.By deleting the dependences among conditions,it becomes possible that the branch instruction with high execution frequency may be given high priority.Though the evaluation results are dependent on the compiler optimization level, we got 6-23% speed-up on IA-64.

Also,this paper presents ternary or more search as a method to enhance the performance of switch statements.The result of experiment shows 4% speed-up.

1　はじめに

命令レベル並列化の進んだ計算機では、分岐処理は実行速度には大きな影響を与える。従来3つ以上に枝分れする多重分岐の処理は、2方向分岐の逐次的実行に変換されることが多かった。このような方式は比較条件が多くなると、クリティカルパスが長くなり、命令レベル並列性の活用を制限することがある。Intel社とHP社で次世代の計算機として共同開発したIA-64計算機は、分岐の効率を向上するために、プレディケーション機能[1, 3, 5]を備えている。並列分岐処理装置とプレディケーション機能を持つ計算機では、複数の分岐処理装置を有効に利用することによって、従来手法よりも効率よく多重分岐を処理することができる。

本稿では、まず、用語と研究対象である多重分岐の種類を説明する。続いて、考案するスケジューリング方式を説明する。ハイパーブロックに対し、プレディケーション機能と並列分岐処理機能を生かし、多重分岐を並列に実行できるようにするスケジューリング方式を提案する。そして、本方式を実際のコンパイラに実装する内容を記述し、評価の結果を示す。引き続き、caseが多いswitch文向けに本方式を拡張し、他の方式と比較する。

2　用語の説明と多重分岐の分類

2.1　プレディケーション

プレディケーションは、プレディケート修飾に基づき、命令を条件付きで実行することである。プレ

ディケート修飾とは、命令を通常通り実行すべきか否かを決定するプレディケートレジスタを指定することである。命令で指定したプレディケートレジスタが真であれば、命令は実行されるが、そうでなければ、nop のように扱われる。プレディケート・レジスタの値は、さまざまな比較やビット・テスト命令によって設定できる。プレディケート付き実行を利用すれば、分岐を避けることができ、しかもコントロール依存からデータ依存への変更を容易にするため、コンパイラによる最適化が単純化される。

2.2 並列分岐

プレディケーションの適用などでプログラムの基本ブロックを大きくすることができる。この大きな基本ブロックは命令レベルの並列性を生かしやすくするが、この結果として大きくなるブロックの境界に複数の分岐命令が集中し、多重分岐が発生しやすくなる傾向がある [1]。VLIW である IA-64 計算機 [1, 3] は分岐レジスタを 3 つ持ち、それらを全て利用する命令グループもあるので、クロックあたり 3 つの分岐を並列に処理できる。

2.3 ハイパーブロック

ハイパーブロック [2, 4] はいくつかのプレディケート付きの基本ブロックの集合である。入口は一箇所しかなく、出口は複数個あり得る。ハイパーブロックの中には分岐や合流があってもよい。それらは、条件付き命令によって直線的な命令列であるかのように扱われる。ハイパーブロックによる分岐命令の直線化は、ILP(Instruction Level Parallelism) の活用に有効である。

ハイパーブロックの形成

ハイパーブロックの形成にあたり、具体的に以下の 2 ステップを踏む [4]。

1. ハイパーブロックの選択
   1 つの基本ブロックをハイパーブロックに入れるかどうかを判断するのは、3 つの要素を考慮する必要がある。
   (a) 実行頻度
   実行頻度が高いブロックに対して、ハイパーブロックに入れる優先度を高くする。
   (b) ブロックのサイズ
   サイズが小さい基本ブロックはハイパーブロックに入れる優先度を高くする。
   (c) 命令の特性
   関数を呼び出す命令と、メモリアクセス命令がある基本ブロックは優先度を低くする。
   以上の 3 つの要素を総合的に考慮し、いくつかの基本ブロックを選択する。
2. ハイパーブロックの形成
   選択された基本ブロックのうち、さらに以下の 2 条件を満たすブロックを選んで、条件変換により、ハイパーブロックとして形成する。
   (a) 条件 1 : 他のブロックから選択された複数の基本ブロックのどれにもコントロールアークがないこと。
   こういうコントロールアークがある場合、図 1 の B5' のようにテール複製をする必要がある。テール複製とは、このコントロールアークの指す先のブロックを複製することである。
   (b) 条件 2 : これらの基本ブロックの中にネストされた内部ループがないこと。
   この 2 つの条件は、ハイパーブロックの入口が 1 つしかないことを保証する。

図 1 は、ハイパーブロックを形成する過程を示す。

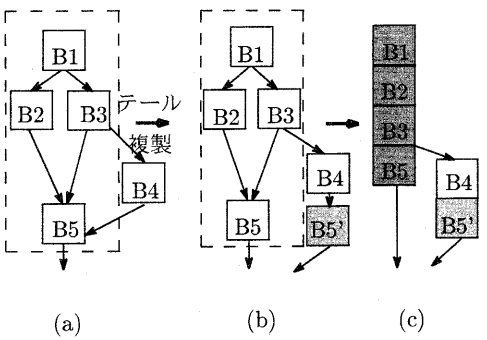

図 1: ハイパーブロックの形成過程 (a) はブロック選択の後、(b) はテール複製の後、(c) は条件変換の後

2.4 多重分岐の種類の分析

多重分岐には、次の 2 種類がある。

1. 例 1 のように、switch 文や switch 文に相当す

るif文は、分岐先の決定を他の分岐条件とは独立に処理できる。以下このタイプを「独立型」の多重分岐と呼ぶことにする。本研究では主にこの種類に対する処理を実装した。

例 1、switch-case 文の場合：

```
int livecar(NODE *n){
      switch(((n)->n_type)){
         case 7:
            vmark(n->value);
            break;       //---(1)
         case 1:
            return(0);   //---(2)
         case 4:
            return(12); //---(3)
         default:
            return(100);//---(4)
      }
}
```

2. もう一種は、例2のように分岐先の決定に複数の条件を判断する必要がある多重分岐である。

例 2、IF 文がネストされる場合：

```
int abc(n){
    if(n==1){
       return(2);       //---(1)
    } else if(n < 10){
       return(3);       //---(2)
    } else if(n < 15){
       return(4);       //---(3)
    } else if(n < 26){
       return(6);       //---(4)
    } else if(n < 70){
       return(8);       //---(5)
    } else if(n < 80){
       return(10);      //---(6)
    }else {
       return(0);
    }
}
```

例1と異なり、(2)に分岐するか否かを判断するには、$n \neq 1$ と $n < 5$ の2つの条件を判断しなければならない。以下このタイプを「非独立型」多重分岐と呼ぶことにする。判断条件が複雑になると、ハードウェアへの要求が厳しくなり、提案する方式を用いても現在のプロセッサでは、効率よく実行できないことが多い。

3 本方式のアルゴリズム

1つの条件分岐を2つの命令(すなわち、比較命令と分岐命令)に分解し、処理する。以下のステップは各分岐先を決定するユニークなプレディケートレジスタを取得するのが目的である。各分岐先に対応するプレディケートレジスタが判別できれば、各分岐命令を並列に実行できる。その後、分岐発生確率の高さの順番でソートし、並列分岐命令をスケジューリングする。記述の便宜のため、プレディケートレジスタをPRと略す。

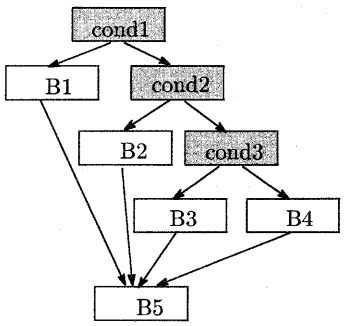

図 2: 従来の依存グラフ

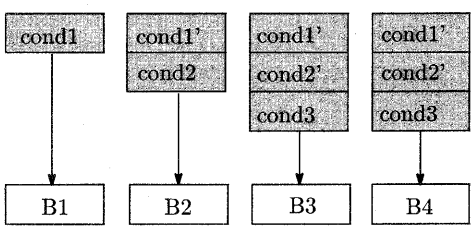

図 3: 本方法による依存グラフイメージ

Step1 比較演算の結果をセットするPRを割り当てる。

1つの比較命令やビット比較命令は、命令で指定した2つのPRに対して比較の結果をそ

れぞれ真値と偽値にセットする。

独立型の場合は、スケジューリング対象の文に関連する比較命令の間で、真値にセットされるレジスタが重ならないように、比較命令に PR を割り当てる。そして、偽値にセットされるレジスタには、p0(書き込まれた値に依らず常に真が読み出せる PR) を割り当てる。

非独立型の場合は、スケジューリング対象の文に関連する比較命令の間で、真値にセットされるレジスタも偽値にセットされるレジスタも重ならないように PR を割り当てる。

Step2 ハイパーブロックを形成する。

2.3 節のハイパーブロックの形成方法に従い、比較命令と分岐命令をハイパーブロックに入れ、分岐先のブロックは入れない。

Step3 比較命令と分岐先の対応表を作り、分岐先別の PR リストを取得する。
このリストの長さは、独立型の場合は 1 となり、非独立型の場合は 1 以上となる。

Step4 分岐先を決めるユニークな PR と、並列実行できる比較命令を取得する。
独立型の場合は、求める PR は対応表にあるレジスタである。
非独立型の場合は、以下のようにする。ある比較命令に先行する比較命令を**親比較命令**、後続の比較命令を**子比較命令**と呼ぶ。図 2 では、cond1 は cond2 と cond3 の親比較命令であり、逆に、cond2 と cond3 は cond1 の子比較命令である。同様に、cond2 も cond3 の親比較命令である。親比較命令を子比較命令の数だけ複製し、並列比較機能を用いて、依存関係がある親子比較命令を同時に実行できるようにスケジューリングする。

非独立型の多重分岐に対して、本方式を適用する前後の依存グラフを、それぞれ図 2 と図 3 に示す。

4 実装

本方式を我々の研究室も開発に参加している COINS (COmpiler INfra Structure) C コンパイラ [7] に実装した。原理的には、並列分岐と並列比較命令を備えていれば、提案する方式を実装可能である。しかし、最新の VLIW 計算機である IA-64 マシンでも、並列分岐装置はあるが、並列比較命令はごく一部の場合 [1] しかサポートされていないのが現状である。このハードウェアの制限で、現在は独立型の多重分岐だけをこの方法で解決する形で実際のコンパイラに実装した。将来、より拡張された並列比較機能を有する計算機が登場したら、本方式をより積極的に適用できるであろう。

実装内容は、以下である。

- プレディケーション実装
  COINS の実装には、if-変換等のプレディケーションを支援する要素が実装されていなかったので、実装した。
- 本スケジューリング方式の実装
  1. 中間言語段階の処理
     3 章のアルゴリズムを実装した。
  2. 依存グラフの処理
     プレディケーションを導入することによって、データフローが変わってしまうが、依存関係が大きく変わらないようにして処理した。
  3. コードジェネレータ段階の処理
     中間言語段階で追加した部分を改造した。

5 評価

5.1 独立型多重分岐の評価

COINS の IA-64 向けコードジェネレータが未成熟なため、生成されたコードでは、速度計測がまだできないので、2.4 節の例 1 のプログラムに対して、インテルコンパイラ ecc と GNU コンパイラ gcc の生成したコードに、本方式を適用し、評価した。

テストプログラムでは、この関数を十万回繰り返し、実行 CPU サイクルをプロセッサのパフォーマンスモニタ機能を用いて測定した。そして、10 回の試行の平均値を表 1 にまとめた。

図 4 と図 5 は、それぞれ ecc コードについて、本方式を適用する前後のコード付き依存グラフである。これらの図は、本研究で開発したツールで作成した。本方式を適用すると、gcc と ecc ではそれぞれ 1 つクリティカルパスを短縮でき、6.1%と 23%の性能向上を確認できた。

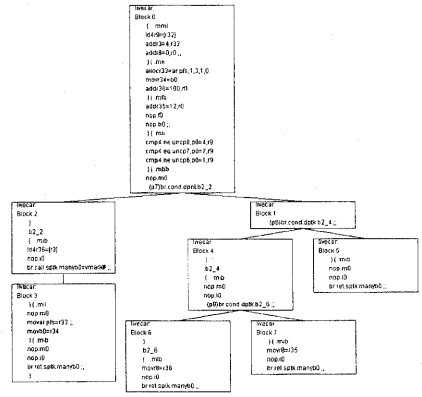

図 4: 例 1 の ecc コードのコード付き依存グラフ

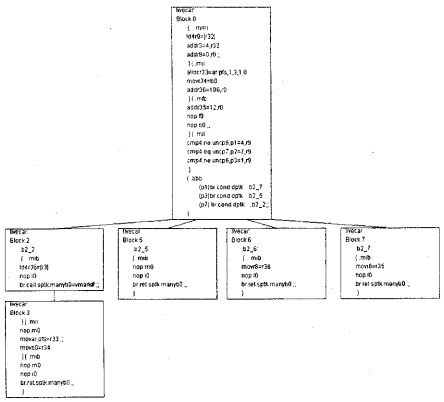

図 5: 例 1 の本方式を適用したのコード付き依存グラフ

表 1: 例 1 の平均実行時間の比較

	CPU サイクル (10^3)	改善比率
ecc	7,105	-
本方式	5,124	23%
gcc	53,514	-
本方式	50,211	6.1%

表 2: 例 2 の実行クロック数の比較

	クロック	改善比率
ecc	7	-
本方式	6	11.7%

5.2 非独立型多重分岐の評価

本方式を非独立型多重分岐に適用するあたり、並列比較命令を利用する必要がある。前述のように、実マシンには、本方式を実装するのが難しい。ここで、他の機能が IA − 64 計算機と同様 (1 クロックで分岐命令が 3 つ、分岐命令以外の命令が 6 実行できる) であり、並列比較機能だけを仮定する仮想マシンで本方式を評価する。結果は表 2 に示す。

2.4 節の例 2 のソースを例題として用いる。

本方式では、親比較命令を複製するので、コードの量が増える。条件個数が 1 個増えば、コードを n(条件の個数) 増やす。d_1 と d_2 でそれぞれ並列分岐装置とそれ以外の装置の個数を表すとすると、下記の不等式が成立する時、本方式が有利である。

$$n/d_1 + n(n+1)/(2 * d_2) < n$$

d_1 と d_2 がそれぞれ 3 と 6(仮想マシンの数値) の場合、上式を解けば、$n < 7$ の時、本方式ではよりよい効果が得られる。

原理的には、抽出した並列実行命令がすべて並列実行できる (並列実行装置の個数が無限である) と仮定すれば、条件分岐を処理するには、本方式では 2 クロックであるが、従来の方式では n(比較条件の個数) クロックが必要である。将来、並列実行装置が多くなればなるほど、本方式の性能を向上できる。

6 n 分探索法への拡張と評価

線形サーチ、ジャンプテーブル及びバイナリサーチは switch 文に対応する分岐処理の代表的な手法である。n の値は分岐関連のハードウェアの資源に依存するが、case の処理を n 分探索に帰着できる。現在の実装対象である IA-64 には並列分岐装置が 3 つあるため、これを有効に利用できるように、本方式を拡張する形である 3 分探索法による case の処理を考案し、試した。

case の個数が多くなる (理論的には 9 以上である) と、3 分木探索法を適用する。本方式と線形サーチの比較は、5.1 節で行ったが、ここで case の個数が大きい場合について、3 分探索法をそれぞれ線形サーチとジャンプテーブルと比較をする。用いたのは、ジャンプテーブルが適用される規模の例題で、case 間の値が不等間隔 (3〜7) であり、case の個数が 21 である。表 3 は、本方式とジャンプテーブルを比較

した結果である。case 文の比較定数が 1,2,3, … のように、多数の連続した数となっているような場合は、ジャンプテーブルの方式をそのまま適用すべきである。

表 4 は、本方式と線形サーチを比較した結果である。上記の例題を用いているが、case の個数を 17 個にした。

本方式は直接バイナリサーチと比較していないが、バイナリサーチの場合、葉までの長さが $\log_2 n$ となるのに対して、本方式は $\log_3 n$ である。

表 3: ジャンプテーブル方式との比較 (case の個数:21)

	CPU サイクル (10^9)	改善比率
ジャンプテーブル	1.86	-
本方式	1.80	4.6%

表 4: 線形サーチ方式との比較 (case の個数:17)

	CPU サイクル (10^9)	改善比率
線形サーチ	2.02	-
本方式	1.74	14%

本方式と switch 文コードジェネレーションの 3 つの主な手法の比較した結果を表 5 に示す。「条件の数」は、case の個数を指す。「case の値域」は、case の値の範囲である。

表 5: case 文に対応する分岐処理の比較

	条件の数	case の値域
線形サーチ	少数	広
ジャンプテーブル	多数	狭
バイナリサーチ	中数	広
本方式	少数〜中数	広

7 おわりに

本稿では、多重分岐を各条件間依存関係の有無にそれほど深くよらず、並列分岐ブロックを抽出することにより、ILP を向上できることと、実行速度が改善できることを示した。今後の課題は、ネストした if 文を処理するとき、増大するコード量を抑えることである。

謝辞

本研究は科学技術振興調整費「並列化コンパイラ向けインフラストラクチャの研究」の補助を受けた。

参考文献

[1] 池井　満: “IA-64 プロセッサ基本講座”, オーム社開発局,2000.

[2] 中田　育男 : “コンパイラの構成と最適化”, 朝倉書店,1999.

[3] Intel Corporation: “Intel IA-64 Architecture Software Developer's Manual,Vol1: IA-64 System Architecture”,Revision 1-4,Document Number: 245317-002〜245320-002,July 2000.

[4] Mahlke S,A.,Lin D.C.,Chen W.Y., Hank R.E.and Bringmann R.A: “Effective Compiler Support for Predicated Execution Using the Hyperblock”, Proc.25th International Symposium and Workshop on Microarchitecture(MICRO-25),pp.45-54,1992.

[5] Carole Dulong,Rakesh Krishnaiyer, Dattatraya Kulkarni,Danie lLavery, Wei Li,John Ng,David Sehr: “An Overview of the Intel IA-64 Compiler”, Intel Technology Journal 4th quarter 1999.

[6] Joseph C.H.Park,Mike Schlansker: “On Predicated Execution” , Software and Systems Laboratory HPL-91-58 May,1991.

[7] 並列化コンパイラ向け共通インフラストラクチャ : http://www.coins-project.org.