計算機アーキテクチャ 147－18
ハイパフォーマンス 89－18
コンピューティング
（２００２．３．８）

述語付き命令を持つ計算機における条件変換の静的最適化方式

朴 小林　鈴木 貢　渡邊 坦

電気通信大学　情報工学科

述語付き命令（predicated instruction）を持つ VLIW 計算機では、分岐を削減することができる。それを用いた条件変換（if-conversion）を行うと、ソフトウェアパイプライニングの可能性も増えて、プログラムの並列性を高めることができる。

述語付き命令を利用すると、分岐予測ミスを無くすことができる。しかしこの代償として、分岐では実行される命令だけをフェッチしてデコードするのに対して、述語付き命令では実行されない命令もフェッチしてデコードする必要が生じる。一方、分岐を利用すると、分岐予測ミスなどの原因で、必ずしも実行効率を高めるとは限らない。分岐予測ミスによるペナルティはプロセッサの高クロック化に伴うパイプライン段数の増加によりますます大きくなっている。

そこで、本論文では、分岐と述語付き命令のバランスをよくし、効率のよいスケジューリングをする方法を提案し、その効果を実験で確認した。

A static optimization of if-convertion on computers with predicated instructions

Boku Sholin, Suzuki Mitsugu and Watanabe Tan

Computer Science, University of Electro-Communications

In VLIW computer with predicated instructions, we can decrease branches by if-conversion. If-conversion is a transformation which converts control dependencies into data dependencies. But predicated instruction machines fetch instructions which will not be executed. How to take balance between using branches and predicated instructions is a problem. In this paper, we give an algorithm and a formula that calculate the balance between using branch and using predication, and we measured execution time of codes with and without if-conversion, and verified the correctness of our algorithm and formula.

1　はじめに

命令レベルの並列性（ILP = Instruction Level Parallelism）を持つ計算機では、プロセッサは同じサイクルに複数の命令を発行できる。命令レベルの並列性は、静的な並列性（コンパイル時にコンパイラが並列性を検出）と動的な並列性（実行時にプロセッサが並列性を検出）に分けることができる。

命令レベル並列プロセッサにおいて、その並列性を最大限に生かすコードを生成するためには、コンパイラによる最適化技術が不可欠であり、より多くの命令レベル並列性を自動検出しなければならない。例えば、IA-64 のような明示的並列命令計算（EPIC= Explicitly Parallel Instruction Computing）のアーキテクチャでは、コンパイラは分岐の前後でコードを移動して、並列性を探さなければならない。しかし、この判断を下すには、コンパイラがコンパイル時に分岐の方向を予測しなければならない。それには、コンパイル前にプログラムを実行してその動作に関する情報を収集するプロファイル・フィードバック方法もあるが、コンパイラで静的に予測する方法もある。

述語付き命令とは、条件判定の結果によって、述語の値が真のときだけ実行し、偽のときは NOP 操作にする命令である。これによって、コンパイラは分岐をなくすことができる。これは、Condition Code をより汎用化したものである。

条件変換は、条件判定の結果によって then 部か else 部が実行される制御依存のプログラムを、述語付き命令を使った直線的なプログラム（真か偽かのデータに依存するプログラム）に変換する方法で

ある。

述語付き命令によって条件分岐が削除されれば、分岐のないループとしてスケジュールできる。述語のついた命令も独立にスケジュールできるため、より効率よくスケジュール可能となる。

ところが、述語付き命令は、実行されない命令を何も実行しないのではなく、実行した結果をレジスタやメモリに書き込むのを止めることによって実現されている [6]。そのため、実行されないはずの命令にも時間がかかってしまう。

本研究では静的分岐予測 [2] を基にして、コンパイル時に分岐をとるかもしくはプレディケートをとるかをきめる方式を提案する。これによりソフトウェアパイプライニングの可能性も増えて、述語付き命令を活用したスケジューリングが可能になり、プログラムの全体実行効率を向上させることができる。

2 VLIW

コンパイラによってスケジューリングを行うなら、並列性を見つけるためのハードウェアは簡素化できる。VLIW（Very Long Instruction Word）方式ではコンパイラが並列に実行できる命令列を発見し、それらを1つの長い命令にまとめる。これによりプロセッサはその長い命令をそのまま実行するだけで済み、ハードウェアによる並列性検出を省略できる。

例えば、IA-64 では図1のようになっている。

3 関連研究

3.1 静的分岐予測

Ball and Larus[3] は SPEC92 等のベンチマークジョブからいくつかの経験則 (heuristics) を見出した。

Wu and Larus の静的分岐予測 [2] では、その結果に対して次のような検討を加え、表1のような結果を得た。まず複数の予測因子の複合確率で局所 CFG（Control Flow Graph）の分岐確率を予測し、その結果を用いて局所的な基本ブロックの実行度数を推定し、それをもとに大域的な実行度数を求める。そしてその実験結果は、動的プロファイルによる予測に近い成績を示していた。

表1のどの経験則にも当てはまらない条件分岐は分岐するとし、的中率は 50%と見積もる。

3.2 プロファイリングによる分岐予測

プロファイリングは、コンパイラで計測機能をプログラムに埋め込み、典型的な入力データのもとで、分岐確率（実行時に分かる情報）等の動作状況を調べる方法である。

しかし、分岐確率は入力データに依存しているので、予測に用いた入力データと本番の入力データで同じ特性が得られるとは限らない。また、大きいプログラムに対するプロファイリングは、そんなに簡単ではない。

4 新方式による分岐の最適化

分岐の最適化の中で、分岐をプレディケーションに置き換えるかどうかを判断する際に指標となる主要な項目は、分岐の選択肢となるブロックのステップ数と分岐予測が的中する確率の二つである。

全部分岐する方法をとると、分岐のオーバーヘッドのため、実行効率が低下する。全部プレディケーションを行う方法をとると、無効な命令で、CPU の命令キャッシュや実行ユニットが溢れるなどの理由で、実行効率が低下する。そこで、分岐とプレディケーションのバランスを次のようにしてとることを提案する。

4.1 新方式の全体の流れ

本手法の全体の流れは次の通りである：

Step 1 静的分岐予測をする。（論文 [2] 参照）。

Step 2 予測サイクル数を求める公式を使って、分岐をとるか、プレディケションをとるかを決める。

Step 3 ソフトウェアパイプライングを含む命令スケジューリングを行う。

4.2 予測サイクル数の算出公式

以下では予想サイクル数を求める公式を導出するが、そこで使うパラメタは、次の通りである。

- 並列度 p：

  1クロックあたり並列に実行される平均命令数、これは対象とするプログラムと計算機に

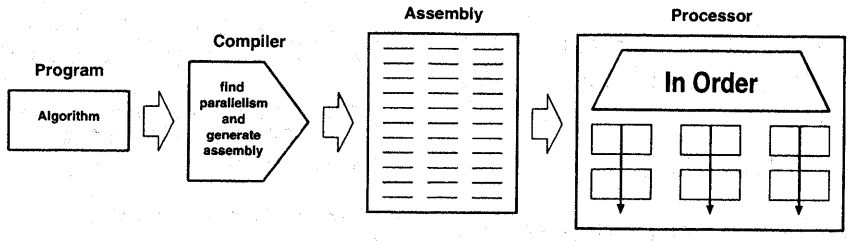

図 1: VLIW 方式による並列実行

経験則	分岐の条件 あるいは 分岐のターゲット	予想	的中率 (%)
Loop	ループヘッダへの戻り	分岐	88
Pointer	ポインタと null またはポインタの比較	非分岐	60
Call	分岐によって支配されない call を含む基本ブロック	非分岐	78
OpCode	$x < 0, x \leq 0, x \equiv c$ の判定 (x:整数変数 c:定数)	非分岐	84
Loop exit	ループからの脱出	非分岐	80
Return	return を含む基本ブロック	非分岐	72
Store	分岐によって支配されない store を含む基本ブロック	非分岐	55
Loop header	分岐によって支配されないループヘッダやループプリヘッダ	分岐	75
Guard	分岐先で設定される前に参照されるレジスタが比較の対象で、分岐によって支配されない場合	分岐	62

表 1: Ball, Wu and Larus の分岐についての経験則とその的中率

よって異なるが、Itanium を計算機モデルとして考えると、2 バンドル同時実行可能で、1 バンドル 3 命令なので、最大 6 になる。

- 命令数 i_{then}, i_{else} :

  if 文の then 部と else 部ごとの命令数、より精確には命令ごとのサイクル数を考えるべきであるが、ここでは簡単化して、1 命令 1 サイクルと仮定する。

- 分岐予測ミスのペナルティ k :

  Itanium では 10 段のハードウェアパイプラインなので、最大 10 となる。分岐予測器を持つ計算機の場合は、分岐予測の失敗率をかけた値となる。

- 分岐予測的中率 B_p :

  論文 [2] 方式で算出した静的分岐予測確率である。

まず、then 部への分岐確率を B_p とすると、then 部が選ばれる時は $B_p \geq 0.5$ であり、その時の then 部のサイクル数は

$$\frac{i_{then}}{p} * B_p$$

と予測される。その予測がはずれる確率は $1 - B_p$ であり、予測ミスによるペナルティを考慮すると、else 部へ行った時のサイクル数は

$$(\frac{i_{else}}{p} + k) * (1 - B_p)$$

となる。

確率変数 X が値 X_i をとる確率が $P_i (i = 1, 2, ..., n)$ であるときの X の期待値 $E(X)$ は

$$E(X) = \sum_{i=1}^{n} X_i P_i$$

なので、$Bp \geq 0.5$ で then 部と予測したときのサイクル数の期待値は、

$$\begin{aligned} E_{branch} &= \frac{i_{then}}{p} * B_p + (\frac{i_{else}}{p} + k) * (1 - B_p) \\ &= \frac{i_{then}}{p} * B_p + \frac{i_{else}}{p} * (1 - B_p) \\ &+ k * (1 - B_p) \end{aligned}$$

である。

続いて、予測結果として else 部が選ばれる場合を考える。その時は $(1-B_p) > 0.5$ であり、else 部のサイクル数は

$$\frac{i_{else}}{p} * (1 - B_p)$$

と予測される。予測ミスで then 部に行ったときにペナルティ $k * B_p$ がかかるので、そのときの then 部のサイクル数は

$$(\frac{i_{then}}{p} + k) * B_p$$

となり、サイクル数の期待値は

$$\begin{aligned} E_{branch} &= (\frac{i_{then}}{p} + k) * B_p + \frac{i_{else}}{p} * (1 - B_p) \\ &= \frac{i_{then}}{p} * B_p + k * B_p \\ &+ \frac{i_{else}}{p} * (1 - B_p) \end{aligned}$$

となる。

上記の 2 つの場合を合わせて考えると、予測結果によって then 部または else 部へ分岐する時のサイクル数の期待値 E_{branch} は

$$\begin{aligned} E_{branch} &= \frac{i_{then}}{p} * B_p + \frac{i_{else}}{p} * (1 - B_p) \\ &+ min(B_p, 1 - B_p) * k \end{aligned}$$

となる。

一方、プレディケートをとる場合の予測サイクル数の期待値 $E_{predicate}$ は

$$E_{predicate} = \frac{i_{then} + i_{else}}{p}$$

となる。

この 2 つの期待値を比較して、サイクル数の少ない方を選ぶことにより、分岐をとるかプレディケートをとるかを決める。

4.3 新方式のアルゴリズム

選択のアルゴリズムは次のようになる。

```
CFG := プログラムの制御フローグラフ
LIST : = 空集合
while (W := CFG から LIST に入っていない
  IF-THEN-ELSE 制御を選ぶ) {
  W に対して、分岐をとったコードを生成し、平
  均の実行サイクル予測値 E-branch を求める。

  W に対して、プレディケートをとったコードを
  生成し、平均の実行サイクル予測値
  E-predicate を求める。

  if (E-predicate <= E-branch
  {
    プレディケーションをとるコードを出す。
  }
  else {
    分岐をとるコードを出す。
  }
  W を LIST に入れる。
}
```

5 評価

本方式は、Coins プロジェクト（文部科学省科学技術振興調整費による「並列化コンパイラ向け共通インフラストラクチャの研究」）のコンパイラをベースにして実装する予定であるが、また IA-64 向けのアセンブリコードを出力するまでに至っていないため、実装はできていない。そこで、手動でいくつかのプログラムに対して、アルゴリズムを適用した。

今回の評価で使った計算機の仕様は、表 2 の通りである。いくつかのプログラムに対して、手動でアルゴリズムを適用し、変換した。

Machine:	HITACHI HA8000-ex/880	
CPU:	Itanium 800MHz x 2	
Cache:	L1:	I: 16kB D: 16kB
	L2:	96kB
	L3:	2MB
Memory:	133MHz, 2G(256MB x 8) SDRAM	
OS:	Redhat Rawhide based on Redhat Linux 7.1 kernel 2.4.3-12.h02smp	
Compiler:	gcc version 2.96 20000731 (Red Hat Linux 7.1 2.96-85)	
	ecc Version 5.0.1, Build 20010705	

表 2: 評価環境

5.1 プログラム

図 2 のバブルソートを例として、上記のアルゴリズムを適用した。ここで、基本ブロックを単位とす

る制御フローグラフでは、I1..In はその基本ブロックに命令 I1..In が含まれることを示している。

```
void bubblesort(int n, int a[])
{
  int i, j, k;
  int x;
  k = n - 1;
  while (k >= 0) {
    j = -1;
    for (i = 1; i <= k; i++)
      if (a[i - 1] > a[i]) {
        j = i -1;
        x = a[j];
        a[j] = a[i];
        a[i] = x;
      }
    k = j;
  }
}
```

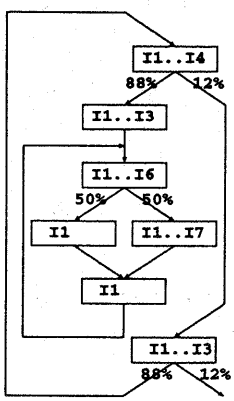

図 2: バブルソート

図 2 の if 文に対するサイクル数の期待値を計算する。k は予備実験から 1 と推定して、並列度 p は 2 とした。

$$
\begin{aligned}
E_{branch} &= \frac{i_{then}}{p} * B_p + \frac{i_{else}}{p} * (1 - B_p) \\
&+ k * min(B_p, 1 - B_p) \\
&= \frac{1}{2} * 0.5 + \frac{7}{2} * 0.5 + 0.5 * 1 \\
&= 2.5 \\
E_{predicate} &= \frac{i_{then} + i_{else}}{p} \\
&= \frac{1+7}{2} = 4
\end{aligned}
$$

これによると、分岐をとるのがはやい。

5.2 実行時間

実際の測定値を表 3 で示した。

測定には次のようなパフォーマンス監視イベントを使った。

- CPU（cpu_cycles）は経過したプロセッサ・サイクル数をカウントする。
- INST（ia64_inst_retired）は、リタイヤした IA-64 命令をすべてカウントする。このカウントには、プレディケート・オンの命令、プレディケート・オフの命令、NOP が含まれる。ただし、ハードウェアにより挿入される RSE 操作は含まれない。

M:$\times 10^6$

プログラム	CPU	INST
predecate	168M	221M
branch	99M	192M
比率	1.7	1.2

表 3: 評価結果

5.3 考察

いろいろなプログラムにおいて、IA-64 における分岐とプレディケートのバランスについて、調べてみた。それで、次のようなことが分かった。

- then 部、else 部の命令数が等しくて、分岐確率が偏っている時に、実行時間が小さい、分岐確率が 0.5 の時に、実行時間が一番大きい。
- 命令数が等しくない場合、命令数が多い方の分岐確率がどんどん低くなっていくことによって、実行時間がどんどん短くなる。述語付き命令を使うと、この場合、時間があまり変わらないと思ったのだが、実際には NOP 操作が短時間で済むため、時間が短くなるのであると思われる。

以上のようにして、本方式を使って得られた予測による判定と実際の測定値に基づく判定が一致することを確認できた。

6 今後の課題

本研究では条件文のコード生成時に分岐にするか、条件変換にするかという問題に対して、実行効率が良いスケジューリングを行うことを目標とした。そしてそれは今回評価したプログラムに関して確認できた。しかし、いろいろの場合について、性能比較することが十分できていない。

以下に残された課題を挙げておく。

- IF 文がネストした場合などを考える必要がある。条件判定を先行実行して、並列化できる命令をどんどん増やして行くことで効率を向上できる。
- ソフトウェアパイプライングによって並列度をあげる。
- プレディケートをとる場合の計算式を改良する。
- 並列度、予測ミスペナルティなどのパラメタの精度をあげる。
- 静的分岐予測を改良する。

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] Aho, A. V., Sethi, R., and Ullman, J. D.: Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.

[2] Wu. Y, Larus, J. R.: Static Branch Frequency and Program Profile Analysis, MICRO'27, Addison-Wesley, pp.1-11(1994).

[3] Ball T., Larus J. R.: Branch Prediction for Free, PLDI'93, pp.300-313(1993).

[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 hypberblock, MICRO'25, Addison-Wesley, pp.1-13(1992).

[5] August, D. I., Hwu, W. W. and Mahlke, S. A.: A framework for balancing control flow and predication, MICRO '30, Addison-Wesley, pp.92-103(1997).

[6] Intel Corporation: Intel IA-64 Architecture Software Developer's Manual, Vol1: IA-64 Application Architecture, Revision 1.1, Document Number:245317-002, July 2000.

[7] Intel Corporation: Intel IA-64 Architecture Software Developer's Manual, Vol2: IA-64 System Architecture, Revision 1.1, Document Number:245318-002, July 2000.

[8] Intel Corporation: Intel IA-64 Architecture Software Developer's Manual, Vol3: Instruction Set Reference, Revision 1.1, Document Number:245319-002, July 2000.

[9] 中田 育男: コンパイラの構成と最適化, 朝倉書店, 1999.

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