計算機アーキテクチャ 142−20
ハイパフォーマンス 85−20
コンピューティング
（2001. 3. 9）

条件分岐を含むソフトウェアパイプライニング

松本岳大　鈴木貢　渡邊坦

電気通信大学　情報工学科

ソフトウェアパイプライニングは，ループからより多くの命令レベル並列性を引き出すための有力なスケジューリング方法である．これを条件分岐を含むループに適用する場合，述語つき命令を利用して分岐のないループに変換するだけでは，分岐先による制限の違いによって無駄が生じる．実行時間が短くなる方を基準にスケジューリングを行い，分岐先で実行される命令だけを展開する方法もあるが，分岐予測ミスによるペナルティが生じる．また，コード量が増大するという問題もある．

本稿では，ループ展開の回数を，分岐先ごとに変える方法を提案する．これは，実行時間を均衡させる形でスケジューリングを行うことにより，分岐先による制限の違いを解消する方法である．本方式は従来法におけるデータ依存グラフの修正によって実現できるため，処理時間も短く，実装も容易である．

Software Pipelining with Conditional Branches

Matsumoto Takehiro, Suzuki Mitsugu and Watanabe Tan

Computer Science, University of Electro-Communications

Software pipelining is an important sheduling technique for extracting instruction level parallelism from a loop. When it is to be applied to a loop with conditional branches, predicated instructions will be useful to eliminate conditional branches from the loop. But this if-conversion is not so effective in case where there is big difference in execution time between then-part and else-part. If inverse-if-conversion is applied, code size is increased and branch prediction miss will cause big penalty.

In this paper, we introduce a new method where the number of loop unrolling differs between then-part and else-part so that the execution time will balance between both parts. The method can be implemented by changing traditional data dependence graph and will not take so much time in compilation.

1　はじめに

コード最適化において，実行時間を短縮するためには，ループの最適化が重要である [1]．命令レベル並列実行の機能を持ったプロセッサ [9] に対しては，ループ内の並列性を引き出すソフトウェアパイプライニング [2] が有効である．しかし，条件分岐を含む場合は，制御依存があるためにソフトウェアパイプライニングが困難になる．

これを意識しないでスケジューリング可能にする方法として，条件文を複合命令とみなす階層的縮約 [3] の方法がある．これでは，1つの複合命令の実行が，ループの繰り返しにまたがることは出来ないため，条件文の実行時間が長いときには効率が悪くなる．

他の方法として，条件分岐そのものを削除する条件変換 [4] の方法がある．この方法では，述語つき命令を使って条件分岐を削除するため，これをそのまま実行できるプロセッサでは，分岐ミスによるペナルティもなくなる．述語つき命令とは，指定された述語 (predicate) の真偽値によって，命令の実行・不実行を制御できる命令のことである．しかし，これを適用するとき，通常は分岐先のうち，制限の強い方を基準にスケジューリングを行うため，分岐先の制限

に差があると，制限の弱い方では効率が悪くなる．制限の弱い方を基準にスケジューリングを行ったとしても [6]，実行されない述語つき命令が無駄になる．この場合は，逆条件変換 [5] の方法によって再び条件分岐のある命令に戻すことが考えられる．この方法では，述語の値に応じて違うプログラムを用意し，それぞれ実行される命令だけを取り出すため，制限の弱い方に分岐したときにも無駄がない．しかし，条件分岐による分岐予測ミスのペナルティが生じる．このペナルティは，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，今後ますます大きくなる可能性がある．また，分岐先ごとのプログラムを用意するために，コード量が増大するという問題もある．

本稿では，述語つき命令によって条件分岐を削除した上で，ループ展開の仕方を場合分けすることにより，実行時間の差による影響を軽減する方法を提案し，その実装，評価の結果を報告する．

2 従来の方法

階層的縮約 [3] では，複合命令の実行を，ループの繰り返しにまたがって行うことができないのに対し，条件変換 [4] では，すべての命令を別々にスケジューリング可能である．

しかし，条件変換では，通常条件文のうち実行時間の長い方を基準にしてしまうので，短い方を実行するときは効率が悪い．

短い方を基準にスケジューリングを行い，長い方はあとから適当な位置に付け足す方法 [6] では，これをそのまま実行しても効果が上がらない．なぜなら，一般に，述語つき命令の機能は，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中止することによって実現されている [7] ため，実行されない命令にも 1 クロックかかってしまうからである．

これに対して逆条件変換 [5] を行えば，分岐先ごとに実行される命令だけを取り出すため，実行されない命令による無駄がない．しかし，分岐ミスのペナルティはかなり大きく，分岐先が予測しにくい場合は無駄が多くなる．

3 本方式の概要

本方式では，分岐先ごとに回数を変えてループ展開をし，実行時間の短い方の分岐先は，長いほうよりも展開する回数を多くする．これにより，実行時間の長い方に分岐するときよりも，より多くの繰り返しが同じ実行時間で実行できるようになり，実行時間の不均衡が解消される．

図 1(a) では B の方が実行時間が長いので，実行時間の短い C の後に次のループ A' を実行するようにループ展開をすると，図 1(b) のようになる．C 側の D の後に次のループの A' を実行し，ここでも C' 側に分岐したときは C' 側の D' の後に A を実行させる．こうすると，C 側への分岐と C' 側への分岐が 1 回ずつ起こる場合，ループの繰り返しが 1 回多く実行できることになる．

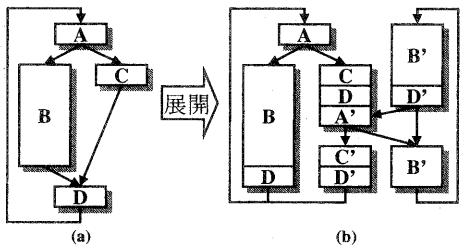

図 1: 本手法の適用例

4 本方式のアルゴリズム

本方式の具体的処理は次のステップからなる．

1. データ依存グラフ (Data Dependence Graph, 以降 **DDG**) を作り，条件変換によって，展開前のループのスケジューリング [8] [10] を行う．

2. ループ展開を行う分岐先を決定し，展開後のループに合わせて DDG を再構築する．

3. 展開後のループのスケジューリングを行う．

4. 命令グループごとに述語を割り当て，述語を設定する命令を追加する．

4.1 DDGの再構築

DDGは，各命令に対応した節を持ち，それらの節を，有向辺で定義から使用へ結んで構成された有向グラフである．

DDGは，$\mathbf{N}$を節，$\mathbf{E}$を有向辺 ($E \subseteq N \times N$)，$\mathbf{CODE}$を命令の集合とすれば，$D \equiv (N, E, L, I)$で定義される．ここで，$\mathbf{L}$は節に対応する命令を返す関数 ($N \rightarrow CODE$)，$\mathbf{I}$は何回の繰り返しにまたがった依存関係かを返す関数 ($E \rightarrow k$)である．

Dをコピーし、これを$\mathbf{D'}(\equiv (N', E', L', I'))$とする展開後のDDGは，Dに D' を加え ($N := N \cup N', E := E \cup E', L := L \cup L', I := I \cup I'$)，有向辺に追加・修正を行うことで得られる．ここで，

- **DEF** は有向辺の定義元を返す関数である．($\langle n \in N, n' \in N \rangle \in E \rightarrow n$)
- **USE** は有向辺の使用先を返す関数である．($\langle n \in N, n' \in N \rangle \in E \rightarrow n'$)
- $\mathbf{E_{back}} \subseteq E$ は帰辺 (実行順と逆の方向への有向辺) の集合である．
- $\mathbf{N_A} \subseteq N$ は展開前ループの条件分岐前の命令 (図1のA)，$\mathbf{N_B}, \mathbf{N_C}, \mathbf{N_D}$ はそれぞれ展開しない分岐先 (図1のB)，展開する分岐先 (図1のC)，分岐先に共通して実行される合流部 (図1のD) の集合である．また，$\mathbf{N'_A}, \mathbf{N'_B}, \mathbf{N'_C}, \mathbf{N'_D}$ は展開後の命令の集合である．

先に述べたDDGの再構築は次のように行う．

- 展開前の命令からの有向辺 $\{e \in E \mid DEF(e) \in (N_A \cup N_B \cup N_C \cup N_D)\}$ を以下にしたがって修正する．

  - $I(e) > 1$ のとき

    展開後の同命令への有向辺を追加する．$E := E \cup \{\langle DEF(e), \{x \in N \mid L(Def(e)) = L(x)\}\rangle\}$

  - $I(e) = 1 \wedge DEF(e) \in N_A$ であり、$USE(e) \in N_A \wedge e \in E_{back}$ または $USE(e) \in N_B$ のとき

    eを，展開後の同命令への有向辺に変更する．$e := \langle DEF(e), \{x \in N \mid L(DEF(e)) = L(x)\}\rangle$

  - $I(e) = 1 \wedge DEF(e) \in N_C$ であり、$USE(e) \in N_A$ または $USE(e) \in N_B \wedge e \in E_{back}$ のとき

    eを，展開後の同命令への有向辺に変更する．

  - 上記以外で $I(e) = 1$ かつ $e \notin E_{back}$ のとき

    展開後の同命令への有向辺を追加する．

  - 展開後の命令からの有向辺については，「展開前 (N_A, N_B, N_C, N_D)」と「展開後 (N'_A, N'_B, N'_C, N'_D)」を入れ替え，同様に修正を行う．

図2では，$5 \rightarrow 1$ が修正する有向辺となる．$DEF(5 \rightarrow 1) \in N_C, USE(5 \rightarrow 1) \in N_A$ であるので，この有向辺の使用先を展開後の同命令 ($1'$) に変更する．また，展開後の $5' \rightarrow 1'$ も同様に，使用先を展開前の同命令 (1) に変更する (図3)．

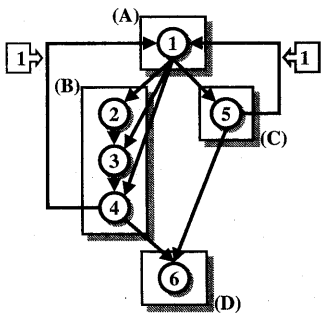

図 2: 展開前のデータ依存グラフ

展開後の命令を実行するかどうかは，展開前の分岐先によって決まる．したがって，展開前

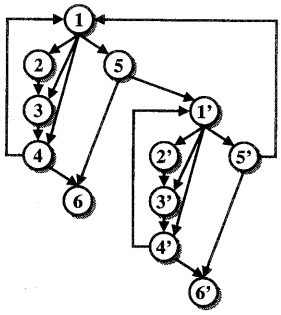

図 3: 展開後のデータ依存グラフ

の分岐先を決定する命令に対応する節から，展開後のすべての節に有向辺を追加する．

4.2 述語の割り当て

述語の割り当てでは，各命令を 4.1 での N_A, N_B, N_C, N_D, N'_A, N'_B, N'_C, N'_D のグループに分けて行う．同一グループ内の命令には同じ述語を割り当て，以下にしたがって述語を設定する命令を追加する．

- N_A では，分岐先を決定する命令で N_B か N_C, N'_A の一方を実行し，もう一方を実行しないように述語を設定する．
- N_B では，N_D を実行し，N'_B, N'_C, N'_D は実行しないように述語を設定する．
- N_C では，N_D を実行するように述語を設定する．
- 展開後の命令については，「展開前 (N_A, N_B, N_C, N_D)」と「展開後 (N'_A, N'_B, N'_C, N'_D)」を入れ替え，同様に述語を設定する

これらのグループは，一部並列に実行されることもあるため，述語の設定は，そのグループが実行される前に行う必要がある．また，直前に設定するようにすれば，述語レジスタの生存区間が短くなり，必要なレジスタ数が少なくなる．N_D を実行する述語の設定を N_A ではなく N_B と N_C で行っているのはこのためである．

5 レジスタリネーミング

本方式では，分岐先を決める命令と，展開したループの間の制御依存が問題になる。展開後の命令が実行されるかどうかは，展開前の分岐先によって決定するため，分岐先決定前に展開後の命令を実行すると，実行されない可能性のある命令を実行してしまうことになる．したがって，展開されたループの実行は，分岐先決定まで待たなくてはならず，分岐先決定に時間のかかるループでは効率が悪くなる．

実行されない可能性のある命令を実行してしまうことによる問題は 2 つある．レジスタやメモリの内容が書き換わってしまうことと，発生しないはずの例外が発生してしまうことである．レジスタの書き換えによる問題は，命令の結果を格納するレジスタに，別の名前をつけることで回避可能である．リネームしたレジスタの値を参照するのは，ループ展開をする方の分岐先なので，展開先の命令で参照するレジスタもリネームする．一方，ループ展開をしない方の分岐先では，これらの命令の結果は使用しないので，そのままにすればよい．その結果，レジスタをリネームされた命令の結果は，参照されることなく捨てられる．

図 4 では，X' の命令の結果を格納するレジスタを，X とは別のレジスタに代え，if',Y' ではそのレジスタを参照するように，変更する．Y の参照するレジスタは変更しないので，X' の結果を参照してしまうことはない．

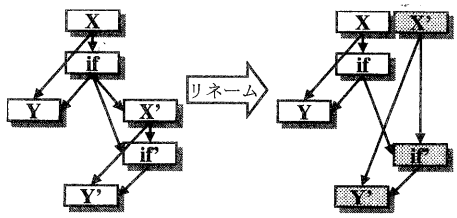

図 4: レジスタリネーム

6 実装したコンパイラの構成

本方式を評価するために，研究室で試作しているコンパイラに対して，以下の機能を追加し，中間コードに対してソフトウェアパイプライニングを行った．

1. 対象ループの選択
2. load/store 最適化
3. データ依存グラフ構築
4. 制御依存を越えたコードの移動
5. 循環路の発見と実行時間の算出
6. モジュロスケジューリング
7. ループ展開とデータ依存グラフ再構築

7 性能評価

整列アルゴリズムのバブルソートに対して本手法を適用した結果から，ループ1回当たりの実行サイクル数を表1に示す．

平均の実行時間では，1000個の乱数を使って実行して分岐先の実行回数を数えたところ，分岐比率はほぼ1:1になった．また，分岐予測ミス確率は，予測のアルゴリズムによって異なるが，今回は直前の分岐先としたところ，予測ミスの確率はおよそ1/3となった．1回の予測ミスによるペナルティを10クロックと仮定すれば，平均約3.3クロックのペナルティとなる．最悪の場合は数列が逆順に並んでいた場合で，すべて実行時間の長いほうに分岐する．最良の場合はすでにソート済みの場合で，すべて短い方に分岐する．

	条件変換	逆条件変換	本方式
最悪	10	10	10
最良	10	2	5
平均	10	9.3	7.5

表 1: 実行時間の比較

8 考察

今回実装した処理では，実行時間の短い分岐先に続いて，次の繰り返しの命令列を展開することを実現した．述語つき命令と命令レベル並列実行の機能を持つVLIW方式のプロセッサ [7] では，命令の空きスロットに述語つき命令を埋め込んでも，命令数はあまり増加しない．この結果，バブルソートのプログラムに対しては，コード量をあまり増大させることなく実行時間が25%向上したことを確認できた．

しかし，現段階では展開可能なループが限定されてしまうため，より多くのプログラムで本方式を適用可能にすることが望まれる．そのために，今後の課題として以下のものが残されている．

- ループ構造による制限の緩和

  現在実装されている処理では，対象とするループの構造を，図1(a)のような形に限定している．さまざまなループ構造に対応するためには，あらかじめループの構造を変換するなどの処理を追加する必要がある．

- レジスタリネーミングによる依存関係短縮の制御

  5章でのレジスタリネーミングでは，命令数が増えてしまう場合もある．現在は，命令数の増大を抑えるために，命令数が増えるときはリネーミングを行っていない．そこで，リネーミングによって性能が向上するかどうかを解析し，必要ならリネーミングを行うような処理が必要になる．

- 複数回ループ展開時の制御方法の検討

  現在は，実行時間の短い方の分岐先に，1回だけループ展開を行っている．これを複数回のループ展開を行うように拡張できるが，その場合，述語の制御が複雑になるため，制御方法を検討する必要がある．

参考文献

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

[2] 中田育男: "コンパイラの構成と最適化", 朝倉書店,1999.

[3] Lam, M.S.: "Software Pipelininig: An Effective Scheduling Technique for VLIW Machines", PLDI'88,pp.318-328,1988.

[4] Dehnert,J.C.,Hsu,P.Y.,Bratt,J.p.: "Overlapped Loop Support in Cydra 5", Proc. 3rd ASPLOS,pp.26-38,1989.

[5] Warter,N.J.,Haab,G.E.,Bockhaus,J.W.: "Enhanced Modulo Scheduling for Loops with Conditional Branches", Proc.IEEE Micro-25,pp.170-179,1992.

[6] 山下義行, 中田育男: "ループ中に条件分岐を含む場合の最適なソフトウェア・パイプライニング", 並列処理シンポジウム JSPP'94,pp.17-24,1994.

[7] Intel Corporation: "IA-64 Applicaton Developer's Architecture Guide", http://developer.intel.com/,1999.

[8] Llosa,J.,Valero,M.,Gonzalez,A.: "Modulo Scheduling with Reduced Register Pressure", IEEE TRANS. ON COMPUTERS,VOL.47,No6,pp.625-638,1998.

[9] Lowney,P.G., Freudenberger,S.M., Karzes,T.J,Lichtenstein, W.D.,Nix,R.P, O'dnnell,J.S.,Ruttenberg,J.C.: "The Multiflow Trace Scheduling Compiler", The Journal of Supercomputing, 7,pp.51-142,1993.

[10] Soo-Mook-Moon,Kemal Ebicioglu: "Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining": ACM Trans. Programming Lang. and Systems,Vol.19,No.6,pp.853-898,1997.