計算機アーキテクチャ 110－8
設 計 自 動 化 73－8
（1995．1．19）

コンディションベクタを用いたコンパイラの最適化

井上 昭彦* 赤星 博輝** 冨山宏之** 若林 一敏*** 安浦 寛人**

* 九州大学工学部情報工学科
** 九州大学大学院総合理工学研究科情報システム学専攻
*** NEC C&C 研究所

本稿では，高位合成 (High Level Synthesis) で用いられている様々なコードモーションを可能とするコンディションベクタを用い，命令レベル並列プロセッサのコンパイラにおける条件文に対する新しいスケジューリング手法を提案する．コンディションベクタは実行条件をビットベクトルで表現したものである．本手法は，コンディションベクタをプログラムの各演算に対して与えることにより，制御構造を意識せずに基本ブロックを越えた広域コード移動を行うことを可能とする．更に，演算間のデータフロー関係にコンディションベクタを与え，スケジューリングと同時にレジスタ割り当てを行うことにより，スケジューリングに適したレジスタ割り当てが可能となる．

キーワード：コンディションベクタ，コードスケジューリング，広域コード移動，レジスタ割り当て

Code Optimization Using Condition Vectors

Akihiko Inoue * Hiroki Akaboshi ** Hiroyuki Tomiyama **
Kazutoshi Wakabayashi *** Hiroto Yasuura **

* Department of Computer Science and Communication Engineering, Kyushu University
** Department of Information Systems, Interdisciplinary Graduate School of Engineering Sciences, Kyushu University
*** C&C Research Laboratory, NEC Corporation

This paper proposes a new code scheduling method for compilers of fine grain parallel processors based on Condition Vectors.

Condition Vector represents the execution condition of each code in a vector form. Each operation attached Condition Vectors can be moved beyond boundaries of basic blocks without considering control structures. Furthermore, we can derive an efficient allocation of registers suitable for the scheduling, simultaneously .

key word : Condition Vector, code scheduling, global code motion, register assignment

1 はじめに

並列処理を行うプロセッサの普及に伴い，その性能を最大限に引き出す最適化コンパイラの開発は重要な課題である．命令レベルの並列性を抽出し，命令の並び換えを行うコードスケジューリングは，非常に重要な最適化技術の1つである．

本稿ではコンディションベクタ (CV: Condition Vector) を用いた，条件文に対する新しい最適化手法を提案する．コンディションベクタのアイデアは高位合成 (HLS:High Level Synthesis) におけるスケジューリング問題に対して若林らが提案したものであり，その有効性が示されている [1][2][3]．筆者らはコンディションベクタをソフトウェアのコンパイラにおける最適化に適用する．各演算に対してコンディションベクタというビットベクトルを与えることにより，制御構造を意識せずに，広域コード移動を行うことが可能である．更に，演算間のデータフロー関係に対して CV を与えることにより，レジスタ割り当ても同時に行うことが可能となる．

本稿では 2 章において従来のコードスケジューリング手法とその問題点について述べる．3 章でコンディションベクタを用いた簡単なスケジューリング例を示した後，4 章でスケジューリング手法の詳細を説明する．5 章で考察を行う．

2 関連研究

一般にコードスケジューリングとは，プログラムのセマンティクスを変えずに，実行ステップ数が最小となるようにアセンブラコードを並び換えることをいう．コードスケジューリングは一度にスケジューリングを適用する範囲により局所コードスケジューリングと広域コードスケジューリングの 2 つに大別される．局所コードスケジューリングにおいては 1 つの基本ブロック内でのみコード移動を行うが，広域コードスケジューリングにおいては基本ブロックを越えたコード移動を行う．

局所スケジューリングアルゴリズムとしては，リスト・スケジューリングが広く用いられている．しかし，スケジューリング範囲が基本ブロック内に限定されているため，基本ブロック内の命令の並列性が低い場合のスケジューリングの効果は低い．

命令レベルの並列度を上げるためには，広域コードスケジューリングを行うことは有効であり，そのアルゴリズムとしてはトレース・スケジューリング [4]，パーコレーション・スケジューリング [5] などが提案されている．トレース・スケジューリングは複数の基本ブロックをあたかも 1 個のトレースと呼ぶ基本ブロックとみなし，各トレースについて広域コード移動を施すものである．しかし，分岐方向に偏りがない条件文の場合，スケジューリングの効果は低い．パーコレーション・スケジューリングは，基本ブロックをノードとし，基本ブロック間の制御の流れをツリーで表したフロー・グラフの下流から上流に向かって 4 種類の基本変換を用いてコードを移動させ，上流の基本ブロックの並列度を向上させる手法である．しかし，全ての実行パスで結果が良くなることを保証してはいない．

ループに関してはループアンローリングやソフトウェアパイプライニングといった手法が効果的であることは示されているが，これらの手法は，もともと並列性の低い条件文の並列性を向上させるものではない．

今回提案する手法は，条件文の並列性を比較的容易に高めることを可能にする．CV によって制御構造を表現し，命令間のデータフロー関係と CV によって“どの演算をいつ実行するか”を決定する．

次章で簡単な例を挙げて，CV の性質および CV を用いたスケジューリング法の説明を行う．

3 コンディションベクタを用いたスケジューリング例

図 1に示すプログラムに対するスケジューリングについて考える．ターゲットとして ALU を 2 個並列に使用できるスーパスカラアーキテクチャを想定する．ALU では加算，減算，乗算，比較命令が実行可能であり，また，簡単のため分岐命令およびジャンプ命令は考えないものとする．

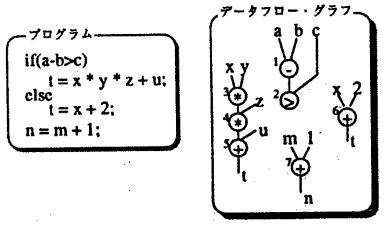
図 1(a) に示すプログラムに対して局所スケジューリングのみを施した場合，最長パスの実行ステップ数は 6 ステップとなる．しかし，図 1(a) に示すプログラムは，広域コード移動を行うことにより 3 ステップで実行され得る (図 1(b) 参照)．

この様な広域コード移動は CV を用いることにより簡単に実現される．基本的なスケジューリングの流れは，

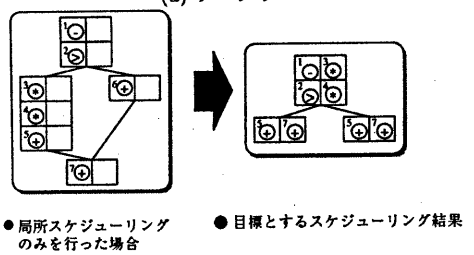
1. プログラム全体のコントロール・データフロー・グラフを作成する．
2. データフロー・グラフの各ノード (演算) に CV を与える．
3. データフロー・グラフと CV を見ながら演算を実行するタイムステップを決定する．

である．

データフロー・グラフの各ノードに対する CV の与え方を図 2を例にとって説明する．まず，プログラム



(a) プログラム



(b) 広域コード移動

図 1: 簡単な例

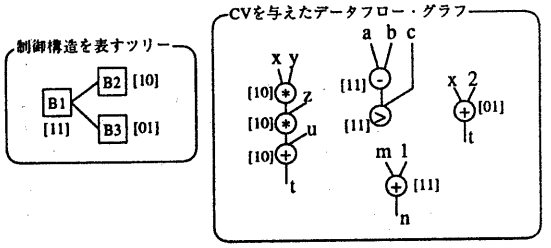


図 2: CV の与え方

	CV	
if(a-b>c)	[11]	[11]
t = x * y * z + u;	[10]	[11]
else		
t = x + 2;	[01]	[11]
n = m + 1;	[11]	[11]

条件 : a-b>c　実行　未実行

図 3: CV の動的な変化

をツリーに展開する．ここでツリーのノードは基本ブロック，エッジはコントロールフロー関係をそれぞれ表す．次に，ツリーの各基本ブロックに CV を与える．ツリーのリーフとなる基本ブロック (図中の $B2, B3$) には 1hot 符号を与え，その他の基本ブロック ($B1$) にはその子ノードになっている基本ブロック ($B2, B3$) の CV の論理和を与える．制御外の基本ブロック ('n=m+1' が存在する基本ブロック) に与える CV は [1]* である．各演算に与える CV は図 2の例のように基本ブロックを越えたデータフロー関係のない場合は，基本ブロックの CV と同じものである (データフロー関係を考慮に入れた CV の与え方の定義は 4.3.3節で行う)．

各演算に対して CV を与えたデータフロー・グラフを図 2に示す．このデータフロー・グラフを用いて演算を実行するタイムステップを決定する．各演算には優先順位を与え，先行制約を満たしかつ優先順位の高い演算から順にタイムステップに割り当てる．

スケジューリングを行っている時，CV は条件演算の割り当てが終了したか否かによって動的に変化する．例えば図 3 で演算'x*y' の CV は，条件演算'a +b<c' の割り当てが終了する前には [11]，割り当てが終了した後では [10] となる．つまり CV が [11] の演算は条件に関係なく必ず実行され，CV が [10] の演算は条件によっては実行されないことを示している．CV の動的な変化を図 3に示す．

*全ての要素が 1 である CV

図 4に CV を用いたスケジューリング結果を示す．図 4において FUV_{ALU} はそのステップにスケジューリングされた演算の CV の和であり，その最大要素はそのステップで実行される演算が使用する ALU の数を表している．並列に利用可能な ALU は 2 個であるので，FUV_{ALU} の最大要素が 2 を越えないようにスケジューリングを行う．

タイムステップ 3 において演算'm+1' のスケジューリングを行う前の FUV_{ALU} は [11] であり，ALU が 1 個余っていることが分かる．そこで，演算'm+1' の CV[11] を [01] と [10] に分割し，タイムステップ 3 に割り当てる．これは分岐外の演算をコピーして分岐内へ移動したことを示している．

4 CV を用いたスケジューリング手法

4.1 入出力

本稿で提案するスケジューリングの入出力は以下のとおりである．

入力 $G(V_I, E_I, V_B, E_B)$: プログラムの入口と出口となる基本ブロックが各々1 つ存在するプログラムの DAG(Directed Acyclic Graph) 表現 (図 5参照)．

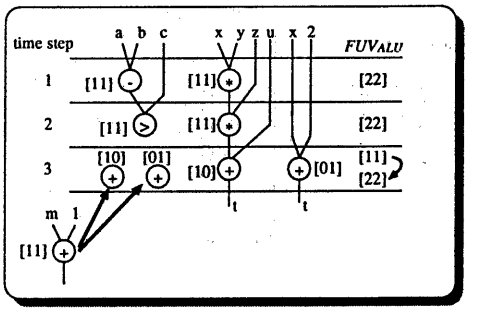


図 4: CV を用いたスケジューリング結果

出力 入力プログラムのセマンティクスを保持した，ターゲットアーキテクチャがもつマシン命令の列

ここで，

- I_i : 1 つのマシン命令により実現される演算
- B_j : 基本ブロック

とすると，

$$
\begin{aligned}
V_I &= \bigcup_{i=1}^{m}\{I_i\} \\
E_I &= \{(I_i, I_{i'}) \mid I_i, I_{i'} \in V_I, \\
&\quad I_i, I_{i'}\text{間にデータフロー関係が存在し，} \\
&\quad I_i\text{が}I_{i'}\text{に先行する}\}
\end{aligned}
$$

である．また，

$$
\begin{aligned}
B_j &\subseteq V_I, \quad \bigcap_{j=1}^{n}\{B_j\} = \phi \\
V_B &= \bigcup_{j=1}^{n}\{B_j\} \\
E_B &= \{(B_j, B_{j'}) \mid B_j, B_{j'} \in V_B, \\
&\quad B_j, B_{j'}\text{間にコントロールフロー関係が} \\
&\quad \text{存在し，}B_j\text{が}B_{j'}\text{に先行する}\}
\end{aligned}
$$

である．

一方，$\forall B_j \in V_B$ に対して $(B_j, B_{j_{in}}) \notin E_B$ を満たす $B_{j_{in}} \in V_B$が唯 1 つ存在し，かつ，$\forall B_j \in V_B$ に対して $(B_{j_{out}}, B_j) \notin E_B$ を満たす $B_{j_{out}} \in V_B$が唯 1 つ存在する．このことは，プログラムの入口，出口となる基本ブロックが各々唯 1 つ存在することを意味する．

4.2 定義

本節では，次節以降で使用する記号の定義を行う．

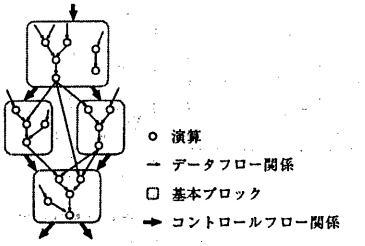


図 5: DAG 表現

- $pred_B(B_j)$: 基本ブロック B_jの先行ブロックの集合．
- $succ_B(B_j)$: 基本ブロック B_jの後続ブロックの集合．
- $pred_I(I_i)$: 演算 I_iの先行演算の集合．
- $succ_I(I_i)$: 演算 I_iの後続演算の集合．
- $block(I_i)$: 演算 I_iが属する基本ブロック．
- $CV(B_j)$: 基本ブロック B_jの CV．
- $cv(I_i)$: 演算 I_iの CV．

4.3 スケジューリング手法

提案するスケジューリング・アルゴリズムの全体を本稿末に記す．

本手法は主として，

1. $G(V_I, E_I, V_B, E_B)$ をより小さな DAG に分割する．
2. 1で分割された DAG をツリーに展開する．
3. DAG の各演算に CV を与える．
4. 上位 (G において入口に近い方) のツリーから順に演算をタイムステップへ割り当てる．必要であれば下位のツリーから演算の移動を行う．
5. 全ての演算が 1 度いずれかのタイムステップへ割り当てられた後，必要であれば上位の演算を下位へ移動する．
6. アセンブラコードを生成する．

の 6 ステップからなる．以降，各ステップの詳細について説明する．

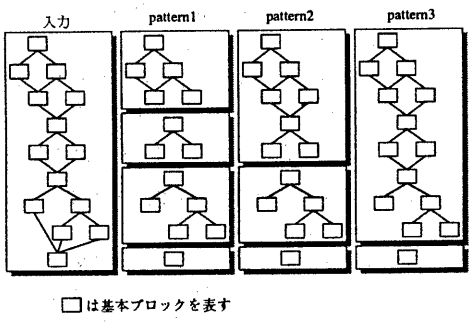


図 6: DAG の分割

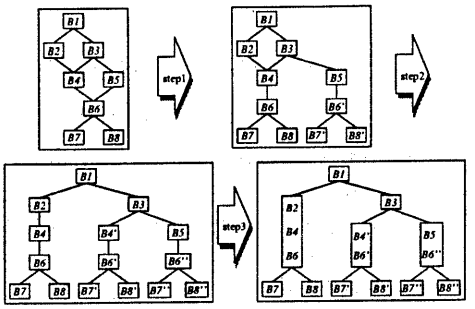


図 7: ツリーへの展開

4.3.1 DAG の分割

分割は DAG の切断点†となる基本ブロック (ただし, ルートノードは除く) で行う. 図 6に例を示す. 図中 (入力) の DAG が与えられた場合, 図に示すように 3 種類の分割の方法がある. どのように分割するかについては, 5章で考察を行う.

4.3.2 ツリーへの展開

CV を適用可能とするため DAG をツリーへ展開する. ツリーへの展開は, $pred(B_j) \geq 2$ である B_j が存在する場合, B_j と $succ(B_j)$ のコピーを作成することによって行う. 例を図 7に示す. 図中で条件 $pred(B_j) \geq 2$ を満たす基本ブロックは $B4$ と $B6$ である. ステップ 1 において $B6$ と $succ(B6)$ のコピーを行っている. 同様にステップ 2 では $B4$ に関してのコピーを行っている. ステップ 3 において, 分岐していない連続した基本ブロック (図中の $B2$ と $B4$ と $B6$, $B4'$ と $B6'$, $B5$ と $B6''$) を 1 つの基本ブロックにまとめる.

4.3.3 CV の計算

ツリー内の演算に CV を与える計算は, 以下の手順で行う.

1. ツリーのリーフとなる基本ブロックの CV として 1hot 符号を与える.
2. リーフ以外のノード (基本ブロック) の CV = そのノードの子の CV の論理和
3. 演算 I_i に与える CV は以下の式で定義する.

$$cv(I_i) = CV(block(I_i)) \quad and\ (cv(I_{i_1})\ or\ cv(I_{i_2})\ or\ \ldots)$$

$$cv(I_{i_1}),\ cv(I_{i_2}),\ \ldots \in succ_I(I_i)$$

4.3.4 上方向へのコード移動

演算のスケジューリングはツリーを単位に, G の上位のツリーから順に行う. ツリーには各々独立したタイムステップを与え, タイムステップ毎に演算の割り当てを行う. ツリー CT におけるタイムステップ $Tstep$ のスケジューリングは以下の手順で行う.

1. CT内の演算の集合を $Irest$, CTより下位のツリーに存在する演算の集合を $Orest$ とする.
2. $Tstep$ において実行可能な $Irest$ および $Orest$ 内の演算の集合をそれぞれ $Iready$, $Oready$とする. ここで, 実行可能な演算 Iとは以下の 2 つの条件を満たすことをいう.
   - 全ての演算 $I' \in pred_I(I)$ が $Tstep$ 以前に割り当てられている.
   - 演算 Iを $Tstep$ に割り当てることによりプログラムの実行結果が変わらない.

   2 番目の条件はデータフロー関係と演算の CV を調べることにより判定することができる.
3. $Iready$が空集合になるまで以下の処理を行う.
   (a) $Iready$内の演算で最も優先度の高い演算を選択する. ここで, 演算 Iの優先度を $priority(I)$ とすると,
   - $I' \in block(I)$ かつ $I' \in succ(I)$ を満たす演算 I'が存在する時,

$$priority(I) = max(priority(I')) + 1$$

†そのノードを除去することにより, グラフが非連結となるようなノード

- 存在しない時，

$$priority(I) = max(priority(I'')) + 1$$

$$I'' \in succ(block(I))$$

(b) 3aで選んだ演算 Iの $Tstep$ における CV を計算する．

(c) 演算 Iが $Tsteps$ に割り当て可能であれば割り当てる．ここで，割り当て可能とは以下の条件を満たすことを指す．

- 演算 Iが使用する演算器を fとすると，$Tstep$ において fを使用する演算の CV の和 FUV_fの最大要素が $Tstep$ で利用可能な演算器 fの個数 R_fを越えてはならない．つまり，
  - $max(CV)$：CV の最大成分
  - $acv(I_i)$：$Tstep$ における演算 I_iの CV

  とすると，

$$max(FUV_f + acv(I_i)) \leq R_f$$

  を満たさなければならない．
- $Tstep$ における全ての演算の CV の和 $FUVall$の最大要素は，$Tstep$ で発行される演算の個数を示している．従って，$FUVall$の最大成分がアーキテクチャの命令発行数 $Rall$を越えてはならない．つまり，

$$max(FUVall + acv(I_i)) \leq Rall$$

  を満たさなければならない．

4. 演算 Iを $Tstep$へ割り当てた時，$Tstep$で必要となるレジスタ数が使用可能なレジスタ数を越えなければ割り当てる．レジスタが使用可能であることは以下の条件を満たすことを指す．

$$max(acv_e(I_i, I_{i'}) + FUVreg) \leq Rreg$$

ここで，$acv_e(I_i, I_{i'})$，$FUVreg$，$Rreg$はそれぞれ，

- $acv_e(I_i, I_{i'})$：エッジ $(I_i, I_{i'})$ の CV
- $FUVreg$：$Tstep$ におけるレジスタの CV の和
- $Rreg$：利用可能なレジスタ数

を表す．レジスタが不足した場合には，以下のどちらかの処理を行う．

- ストア命令を $Tstep$ に割り当て，ロード命令を DAG に挿入する．
- 現在割り当てようとしている演算をあきらめ，他の演算を割り当てることを試みる．

5. $Oready$が空集合になるまで以下の処理を行う．

(a) $Oready$内の演算で最も優先度の高い演算を選択する．

(b) 5aで選んだ演算 Iの CV を分割する．ただし，分割された演算の CV は 1hot 符号でなければならない．分割されたそれぞれの演算 $I_{i'}$について以下の処理を行う．

i. $pred(I_{i'})$ が割り当てられているタイムステップから $Tstep$ までのいずれかのタイムステップに演算 $I_{i'}$が割り当て可能であれば割り当てる．

ii. 割り当て可能でない時，これまで割り当てた $I_{i'}$を全てキャンセルし，Iを $Orest$ へ戻す．

4.3.5 下方向へのコード移動

全ての演算がいずれかのタイムステップへスケジューリングされた後，下位のタイムステップ $Tstep$ から順に以下の処理を行い，$FUVall$が [0] となるタイムステップを可能な限り増やす．

- $Tstep$ に割り当てられた演算のうち，$Tstep$ より遅いタイムステップで実行可能，かつ，割り当て可能であるものは可能な限り遅いタイムステップへ移動する．

4.3.6 アセンブラコード生成

スケジューリングを施した DAG からアセンブラコードを生成する．同一ステップに条件排他な演算の割り当てがある場合，そのステップに存在する CV[1] の演算は条件排他なパスへそれぞれ分割する必要がある．

図 8に例を示す．図 8に示すような入力をスケジューリングした結果，演算 1,2,3 が同一ステップに割り当てられたとする．演算 1 は CV が [11] であるため，条件に関係なく常に実行さなければならない．従って，アセンブラコードを生成する際には演算 3 は [01]，[10] の 2 つのパスに分割される必要がある．生成されたアセンブラコードを同図に示す．

5 考察

本稿で提案したスケジューリング手法は，プログラムの全ての実行パスについて，局所コードスケジュー

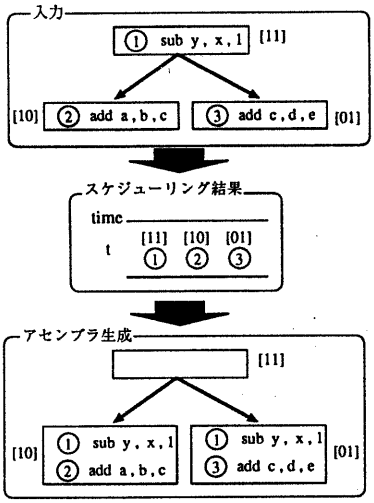


図 8: アセンブラコード生成時における演算の分割

リングのみを施した場合よりも実行時間が短くなる†. このことは，分岐する確率に偏りがない場合，あるいは，分岐する確率をあらかじめ予測できない場合にも，本手法が効果的であることを意味する．また，本手法においては，スケジューリングと同時にレジスタ割り当てを行うため，レジスタ割り当てをスケジューリングに先だって行った場合生じるレジスタ間の依存関係(逆依存，制御依存)を無視することができる．従って，スケジューリングに適したレジスタの割り当てを行うことが可能となる．

本手法では，1つのプログラムのDAG表現を複数の小さなDAGに分割することを4.3.1節で述べた．DAGを少数の大きなDAGに分割すると以下に述べる利点により，実行速度が速くなることが期待される．

- コード移動の機会が増加する．
- ツリーに展開する際に複数の基本ブロックを1つにまとめることができ，ある1つの実行パスを実行したときのジャンプ命令の数が減少する．

しかし，基本ブロックのコピーを多数作成する必要があり，コードサイズは増大する．実行時間とコードサイズとのトレードオフを考慮に入れて，DAGをどのように分割するかを決定しなければならない．

スケジューリングの過程において，ある演算をタイムステップに割り当てようとしたときに，レジスタの不足のため，割り当てることができなかったときの対処法として，1) ロード/ストア命令を挿入する，2) 別の演算を割り当てることを試みるという2つの方法を4.3.4節で挙げた．2) の方法を選択した結果，そのタイムステップに演算を割り当て可能な演算が存在しなかったときは，1) の方法を採らなくてはならない．1) を選択したときは，どの値をメモリにストアするかを決定しなければならない．最も遅いタイムステップで使用されると思われる値をメモリにストアする方が得策である．値が使用される演算の優先度が低いほど，その値は遅く使用されると思われるが，今後深く検討する必要がある．

†最悪の場合でも，局所コードスケジューリングを施した場合と同等となる．

6 おわりに

本稿では条件文に対する新しいスケジューリング手法を提案した．本手法は制御構造を意識せずに広域コード移動を行うことができる．またスケジューリングの過程でレジスタ割り当てを行うため，スケジューリングに適したレジスタ割り当てを行うことが可能である．今後，本手法のインプリメントを行い，本手法の評価を行いたい．

参考文献

[1] K. Wakabayashi and T. Yoshimura, "A Resource sharing and Control Synthesis Method for Conditional Branches," *ICCAD-89*, pp.62-65, Nov 1989.

[2] K. Wakabayashi, "Cyber: High-Level Synthesis System from Software into ASIC," *High-Level VLSI Synthesis*, Kluwer Academic Publishers, June 1991.

[3] K. Wakabayashi and H. Tanaka, "Global Scheduling Independent of Control Dependencies Based on Condition Vectors," *Proc. of 29th DAC*, pp.112-115, June 1992.

[4] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers*, vol.C-30, no.7, pp.478-490, July 1981.

[5] A. Aiken and A. Nicolau, "A Development Environment for Horizontal Microcode," *IEEE Trans. Software Engineering*, vol.14, no.5, pp.584-594, May 1988.

```
main {
    G を n 個の DAG に分割;
    各 DAG をツリー (CT_1, CT_2, ..., CT_k, ..., CT_n) に展開;
    for k := 1 to n do {
        Tstep_k := 0;
        Ijump := CT_k内の無条件分岐 (jump) 命令の集合;
        /* ツリーのリーフでない基本ブロックに jump 命令が存在することはない. */
        Irest := CT_k内の jump 命令以外の命令の集合;
        Orest := CT_{k+1}, CT_{k+2}, ..., CT_{k+n}内の演算の集合;
        forall I_i ∈ Irest に対して, cv(I_i) を計算; /*∀I_i' ∈ Orest, cv(I_i') = [1] */
        while (Irest ≠ φ) {
            Tstep_k := Tstep_k + 1;
            Iready := {Tstep_kにおいて実行可能な Irest 内の演算 }; Irest := Irest − Iready;
            Oready := {Tstep_kにおいて実行可能な Orest 内の演算 }; Orest := Orest − Oready;
            while (Iready ≠ φ) {
                I_i := Iready内で最も優先度が高い演算; Iready := Iready − {I_i};
                acv(I_i) := Tstep_kにおける命令 I_iの CV;
                if (allocatable) then {
                    if (max(acv_e(I_i) + FUVreg) ≤ Rreg) then {
                        I_iを Tstep_kに割り当てる; Iready, Oreadyを更新;
                        FUV_f := FUV_f + acv(I_i); FUVall := FUVall + acv(I_i);
                    } else Load/Store 命令を挿入;
                } else Irest := Irest ∪ {I_i};
            }
            while (Oready ≠ φ) {
                I_i' := Oready内で最も優先度が高い演算; Oready := Oready − {I_i'};
                I_i'を I_i'_1 ... I_i'_m に分割, ただし, cv(I_i') = Σ_{l=1}^{m} {cv(I_i'_l)};
                foreach (I_i'_l) do {
                    for t := (I_i'_lの succ(I_i'_l) が割り当てられている Tstep) to Tstep_k do {
                        if ((allocatable) ∨ (max(acv_e(I_i) + FUVreg) ≤ Rreg) then {
                            FUV_f, FUVallを更新; Oreadyを更新;
                        } else {
                            先に割り当てを行った I_i'_{l-1} から I_i'_1 までをキャンセルし I_i' を Orest へ戻す;
                            Orest := Orest ∪ {I_i'};
                        }
                    }
                }
            }
        }
        Ijump 内の命令 (jump 命令) をスケジュールする;
    }
    for k := n downto 1 do {
        for Tstep' := Tstep_k − 1 downto 1 do {
            演算を可能な限り遅いタイムステップへ割り当てる;
        }
    }
    スケジューリング結果をもとに, アセンブラコードを生成;
}

subroutine allocatable {
    if ((max(FUV_f + acv(I_i)) ≤ R_f) ∨ (max(FUVall + acv(I_i)) ≤ Rall)) then {
        return (TRUE);
    } else   return (FALSE);
}
```