ハイパフォーマンス
コンピューティング 57−21
（1995．8．25）

命令レベル並列プロセッサに対するコンディションベクタを用いた広域コードスケジューリング手法の評価

井上 昭彦* 赤星 博輝* 冨山 宏之* 若林 一敏** 安浦 寛人*

* 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

命令レベル並列プロセッサに対する広域コードスケジューリングは，基本ブロックを越えて演算の移動を行い，プログラムの命令レベル並列性を抽出する．その際，基本ブロックを越えた演算の移動の解析，および，それにともなうプロセッサ資源の管理が複雑になるという問題がある．我々が提案しているコンディションベクタを用いたスケジューリング手法は，演算の移動とプロセッサ資源の管理をコンディションベクタにより一括して行う．本稿では我々が提案したスケジューリング手法のアルゴリズムを述べ，その実験結果を報告する．実験の結果，提案手法の有効性を確認することができた．

キーワード：コンディションベクタ，広域コードスケジューリング，命令レベル並列性，IF 文の並列化

Code Scheduling for Fine Grain Parallel Processors based on Condition Vectors — Preliminary Evaluation Results —

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

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

Global code scheduling enables instructions to move beyond boundaries of basic blocks and exploits instruction parallelism. Global code scheduling has difficulty in analysis of code motion beyond boundaries of basic blocks and resource sharing. We propose a global code scheduling algorithm using condition vectors to resolve these problems. Our algorithm has one of advantages that code motion and resource sharing are performed simultaneously, considering only with condition vectors.

In this paper, our scheduling algorithm is described and the results of experiments prove effectiveness of our algorithm.

Key Words : Condition Vector, Global Code Scheduling, Instruction Level Parallelism, Parallelized Branch

1 はじめに

我々は，コンディションベクタ (CV:Condition Vector) というデータ構造を用いた命令レベル並列処理を行うプロセッサに対する新しい広域コードスケジューリングの方法を提案している [1]．提案したスケジューリング手法 (CVCS:Condition Vector Code Scheduling) はリスト・スケジューリング [2] を広範囲で行うものである．リスト・スケジューリングはその適用範囲が基本ブロックに限られていた．CVCS はその適用範囲を基本ブロックに限らず，基本ブロックを越えた広範囲なリスト・スケジューリングを行う．基本ブロックを越えてスケジューリングを行う場合，基本ブロック外への演算の移動の解析，およびプロセッサ資源の管理が複雑になるという問題が挙げられる．

CVCS では，各演算に CV を与えることにより，演算の基本ブロック外への移動を CV により表現すると共にプロセッサ資源の管理も同時に行う．本稿では，文献 [1] において提案した CVCS を拡張したアルゴリズムを述べ，その評価を行う．

CVCS は命令を並び換える手法と異なり，各演算をその演算が実行されるタイムステップに割当てる手法であるため，スケジューリング結果から一列のアセンブラ命令列を生成する必要がある．アセンブラ命令列を生成する際，実行ステップ数を増加させる JUMP 命令の生成を可能な限り抑えなければならない．本稿では，この問題をグラフの最大マッチングを求める問題に定式化する．

2 章において筆者が提案している CVCS を簡単に説明する．3 章で CVCS を拡張したアルゴリズムを述べると同時に，アセンブラ命令列を生成する際の問題点を挙げ，その問題の定式化を行う．4 章において CVCS の評価を行う．

2 コンディションベクタを用いたスケジューリング (CVCS)

コンディションベクタ (CV:Condition Vector) は演算の実行条件を表現するビットベクトルであり，ベクトルの各要素はある条件の成立，不成立を表している．図 1(b) は (a) のプログラムのコントロールフロー・グラフ (CFG:Control Flow Graph) であり，CFG の各節点は基本ブロックを表している．各文に与えられているビットベクトルが CV である．以下に CV の各要素が表現している事項を示す．

- 上位ビットが 1 → *condition* が成立．
- 下位ビットが 1 → *condition* が不成立．

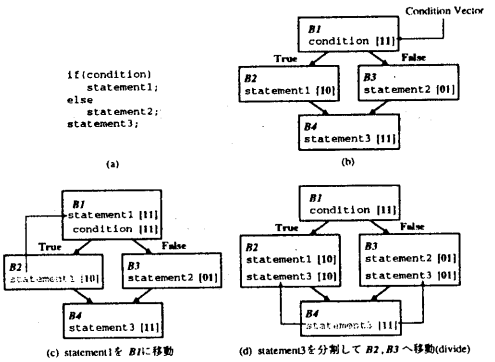

図 1: コンディションベクタ

CV は各演算に対して与えられ，演算の移動に伴いその要素が変化する (図 1(c),(d))．

CVCS はプログラム中の条件分岐文を対象としたスケジューリング手法である．プログラム中にループ文が含まれる場合は最内ループを対象とする．ここで，IF 文 S のレベル $level_S$ を以下に定義する．

- IF 文 S が入れ子になっていない場合
  $level_S = 0$
- IF 文 S が IF 文 T の入れ子になっている場合
  $level_S = level_T + 1$

トップレベル IF 文 (TIF:Top Level IF statement) をレベル 0 の IF 文とする．CVCS はプログラムを TIF 毎に区切り，各 TIF を単位にスケジューリングを行う．スケジューリングは命令を並び換える手法とは異なり，各演算をタイムステップに割当てることにより行う (図 2参照)．

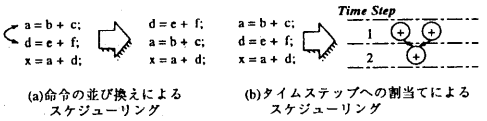

図 2: 演算を並び換える手法とタイムステップに割当てる手法との違い

CVCS はスケジューリングと同時にレジスタ割当を行うことが可能であるが，本稿ではレジスタ割当については考慮に入れないものとする．レジスタ割当をしない CVCS を CVCS-1 とする．TIF_k に対する CVCS-1 のスケジューリングアルゴリズムを図 3に示す．図 3における FUV_f は，あるタイムステップに割当てられている演算器 f を使用する演算の CV の和である．FUV_f の最大値はそのタイムステップで使用される演算器 f の数であ

り，これは1サイクルで使用可能な演算器 f の数 R_f を越えてはならない．

CVCS の詳しいアルゴリズムは [1] を参照して頂きたい．

```
Irest = operations in TIF_k except for jump operations;
Orest = operations in TIF_{k+1}, ...;
Tstep = 0;
while (Irest ≠ φ) {
    compile Iready for Tstep from Irest;
    compile Oready for Tstep from Orest;
    while (Iready ≠ φ) {
        Tstep = Tstep + 1;
        n = operation with largest priority in Iready;
        calculate cv of n in Tstep;
        if(MaxElement(FUV_f) ≤ R_f)then {
            allocate n in Tstep;
            calculate FUV_f;
        }else{
            Irest = Irest ∪ n;
        }
    }
    while (Oready ≠ φ) {
        n = operation with largest priority in Oready;
        divide n;
        if (the schedule becomes better)then {
            allocate the divided nodes in the formerTstep;
        }else {
            Orest = Orest ∪ n;
        }
    }
}
```

図 3: TIF に対する CVCS-1 アルゴリズム

3 CVCS-1 の拡張

CVCS-1 では，ある TIF のスケジューリングが終了した時点で，演算器を有効に使用していないタイムステップが生じる可能性がある．本章では，それらの演算器を有効に使用するために CVCS-1 の拡張を行う．拡張を行った CVCS-1 を CVCS-2 とする．本章ではさらに，アセンブラ命令列を生成する際に生じる問題の定式化も行う．

3.1 トップレベル IF 文の並列化

トップレベル IF 文 (TIF) の並列化とは TIF を単位としてプログラムを区切り，その区切られた各 TIF をオーバーラップしてスケジューリングする手法である．TIF の並列化アルゴリズムを図 4 に示す．TIF の並列化を図 5(i) のプログラムを例に挙げて説明する．図 5(i) の 1 および 7 の IF 文は TIF である．同図 (ii) は (i) に対する CFG であり，TIF1, TIF2 に分割される．TIF1 に対してスケジューリングを施した結果を図 6に示す．1 サイクルで ALU が 2 個使用可能なマシンを想定し，加算，減算，比較演算を ALU で行うとしている．また，全ての演算は 1 サイクルで終了し，1 サイクルで 3 方向以上分岐できない仮定している．簡単のため分岐命令は無視して考えている．

図 6における $REST$は 1 行目 if($t > 0$) においてどちらへ分岐してもタイムステップ 2, 3 において使用されない ALU がそれぞれ 1 つ存在するこ

```
divide a program into TIFs;
Rest_FU_f = number of FU_f;
for(TIF = the first TIF to the last do{
    CVCS-1(TIF);
    Rest_f = Rest_f − number of used FU_f for TIF;
}
```

図 4: TIF の並列化アルゴリズム

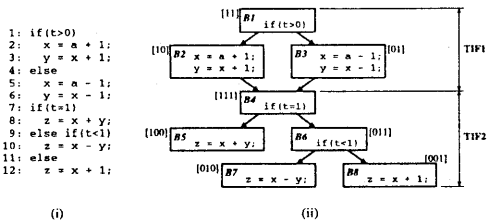

図 5: 例題プログラム

とを示している．この余った ALU を有効に利用するように TIF2 のスケジューリングを行った結果を図 7に示す．演算 $t = 0$, $t < 1$ を TIF1 のタイムステップ 2 および 3 にそれぞれ割当てることにより ALU が有効に利用されていることが図 7 より確認できる．例題プログラムに対してリストスケジューリングを施して実行すると最大 6$step$, 平均*5.7$step$ の実行ステップを必要とする．TIF の並列化をこのプログラムに対して適用することによりプログラムの実行ステップ数を最大 4$step$, 平均 3.3$step$ にまで減少させることが可能となる．

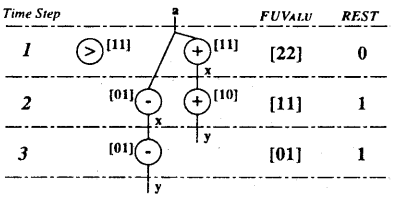

図 6: TIF1 に対するスケジューリング

3.2 スケジューリング結果からのアセンブラ命令列生成

CVCS-2 は演算をタイムステップに割当てることによりスケジューリングを行うため，入力されたプログラムの制御構造 (CFG の形状) を変化させてしまう場合がある．制御構造が変化する場合，スケジューリング結果から直接一列のアセンブラ命令を生成するのは困難である．そのため，以下の手順により CVCS-2 のスケジューリング結果から一列のアセンブラ命令列を生成する．

*平均ステップ数=$\frac{\text{各実行パスのステップ数の総和}}{\text{実行パスの数}}$，ただし全ての実行パスは等確率で実行されるものとする

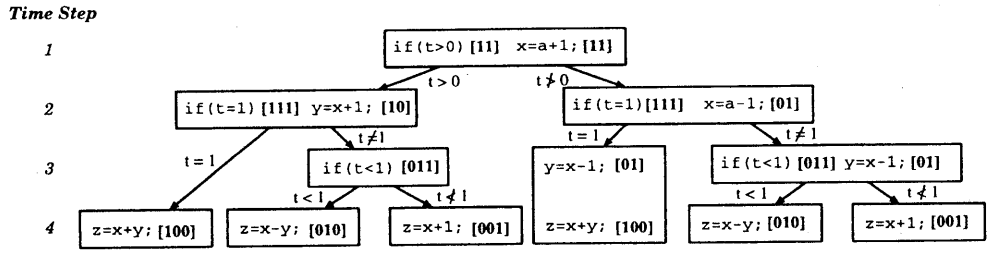

図 8: スケジューリング結果から CFG の生成

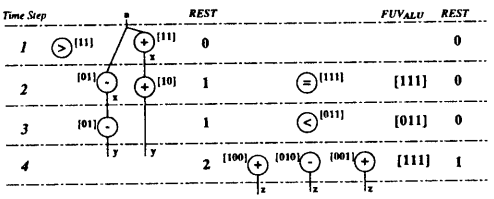

図 7: TIF2 に対するスケジューリング

1. スケジューリング結果から CFG を生成する.
2. 1で生成した CFG の各節点である基本ブロックを一列に並べ, 基本ブロック列を生成する.

以下, 2 つの手順について詳細な説明を行う.

3.2.1 スケジューリング結果からの CFG の生成

スケジューリング結果からの CFG の生成はタイムステップ順に行う. 同一ステップに CV の論理積が [0]† である演算が 2 以上割当てられている場合, それらの演算は異なる基本ブロックに生成される. 図 8は図 7から CFG を生成した結果である. 各演算の右にはその演算の CV を示している. タイムステップ 2 において演算 $y = x + 1$, $x = a - 1$ の CV の論理積は [0] である. したがって, これらの演算は図 8において異なる基本ブロックに生成されてる.

3.2.2 基本ブロック列生成

アセンブラ命令列の生成は CFG の節点である基本ブロックを一列に並べることにより行う. CFG から基本ブロックを一列に並べる際の問題点として, JUMP 命令の挿入により, 生成されたアセンブラ命令列の実行ステップ数が増加する可能性があることが挙げられる. 図 9において CFG(a) に対する基本ブロック列の 2 種類の生成例を (b),(c) に示す. 図中の矢印は JUMP 命令による無条件分岐を表している. (b) の生成例では JUMP 命令を 3 つ必要としており, (c) の生成例に比べ 2 つの JUMP 命令が余分に必要となる. その結果 (b) の実行ステップ数が, スケジューリング前のプログラムの実行ステップ数に比べ増加している可能性がある. スケジューリングを行った結果プログラムの実行ステップ数が増加することは, コンパイラにとって好ましくない. したがって, JUMP 命令の生成を可能な限り抑えるように基本ブロックを一列に並べることが望まれる. 次節において, 一般的な CFG から基本ブロック列を生成する問題をグラフの最大マッチング問題を求める問題として定式化を行う.

†全ての要素が 0 であるビットベクトル

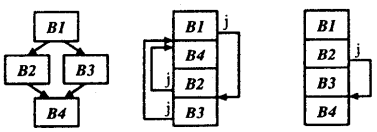

図 9: jump 命令を最小化する基本ブロック列の生成

3.3 JUMP 命令最小化問題

3.2.2節において生じた問題,「**JUMP 命令の生成を可能な限り抑えるように, CFG の節点である基本ブロックを一列に並べよ**」は, 二部グラフの最大マッチングを求める問題に定式化される.

まず, 以下の手順で CFG から二部ラフ $G_{CFG}(V, E)$ を作成する. CFG の各節点をコピーして G_{CFG} の節点集合 V_{in}, V_{out} へ加える. CFG の基本ブロック B_i, B_j 間にコントロールフロー関係が存在するとき, B_i に対応する節点 $v \in V_{out}$ と B_j に対応する節点 $u \in V_{in}$ に枝を接続する (図 10参照).

変換された二部グラフにおいて $v \in V_{out}$ から $u \in V_{in}$ へ枝が存在するとき, 節点 v に対応する基本ブロックの下に節点 u に対応する基本ブロックを並べて生成するものとする. 以上より, CFG から基本ブロック列を生成する問題は二部グラフ

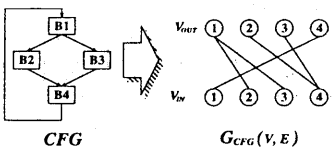

図 10: 二部グラフへの変換

G_{CFG}からある制約を満たす最大マッチングを求める問題として以下のように定式化される.

given CFG を二部グラフへ変換したグラフ $G_{CFG}(V, E)$

problem 以下の制約条件を満たすように二部グラフ $G_{CFG}(V, E)$ の最大マッチングを求める.
[制約条件]
最大マッチングの節点集合において同じラベルをもつ節点間に新たな辺を接続してできた二部グラフに閉路が含まれていない

最大マッチングを求めたとき, 図11に示すように既に生成されている $B1$ を $B4$ の後に生成しなければならないとう矛盾が生じる場合がある. この矛盾を避けるために, 上記の制約条件を満たす最大マッチングを求める必要がある.

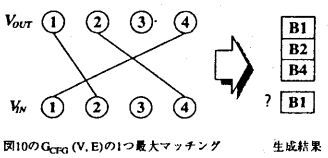

図 11: 問題となる最大マッチング

以下の手順で問題に対する解を求める.

1. 与えられた二部グラフに対する最大マッチングを 1 つ求める.
2. 1で求めた最大マッチングが制約条件を満足するならばそのマッチングが解となる. 制約条件を満足しないなら 1へ戻る.

4 実験

CVCS-2 の評価を行うために簡単な実験を 2 つ行う. 以下の制限を加えたマシンをターゲットとして想定する.

- 以下の機能ユニットを持つ並列度が 4 の VLIW タイプのマシン. ALU ユニット, 乗算ユニット, Load/Sore ユニット, 分岐ユニット.
- 全ての演算は 1 サイクルで実行を終了する.
- 1 サイクルで 3 分岐以上は行わない.
- レジスタ数は十分大きい.

4.1 実験 1

リスト・スケジューリング (LS)[2] および CVCS-2 に関して, スケジューリング後生成されるプログラムの平均ステップ数の比較を行った. ここで, 全ての実行パスは等確率で実行されるものとする. 実験に用いたプログラムを表 1 に示す. それぞれのプログラム中からループ文を含まない関数を選択し, その関数をベンチマーク・プログラムとして用いた. 表中の LKF15 は Livermore Fortran Kernal 15 であり, CV1~3 は, CVCS-2 を実現するプログラムの一部である. 結果を表 2 に示す. 表 3 はターゲットマシンにさらに ALU ユニットを 1 つ加え, 並列度を 5 とした場合の結果である.

CVCS は LS を施した場合と比較して, プログラムの実行ステップ数が 11.4%~85.7%減少した. 実験の結果, プログラム中の全ての実行パスについて, CVCS-2 は LS を施した場合に比べて, 実行ステップ数が短く, または同等となった. CVCS-2 のアルゴリズムは全ての実行パスに対してスケジューリング前に比べ実行ステップ数が増加しないことを保証している. したがって, 分岐方向に偏りがない場合, あるいは, 分岐する確率を予測できない場合にも CVCS-2 は効果的である.

4.2 実験 2

ベンチマーク・プログラムに LFK1 から LFK5 の最内ループを用い, LS, CVCS-2, およびトレース・スケジューリング (TS:Trace Scheduling) [5] を施した後のループ 1 回分の平均実行ステップ数の比較を行った. 分岐確率は条件分岐において成立する方を 0.9, 成立しない方を 0.1 とした. トレースは実行される確率の高い基本ブロックにより構成されるようにした. 結果を表 4 に示す.

表 4 において, 多くの場合 TS を施した方が CVCS-2 に比べ, 実行ステップ数が多くなっている. 演算を基本ブロックを越えて移動する際, CVCS-2 では全ての実行パスのうちどれか 1 つでもステップ数が増加する場合にはその演算の移動を行わない. しかし, TS では他の実行パスに関わらず, 選択したトレースの実行ステップ数が短くなれば, その演算の移動を行う. 実験では, TS を施すことにより分岐後実行される演算の多くがその分岐を決定する分岐命令の前へ移動した. それにより, 分岐する前に多くの実行ステップを要するため, 上記の差が生じたと考えられる.

表 1: ベンチマーク・プログラム (実験 1)

プログラム	quicksort	heapsort	sim	nsieve	LKF 15	CV1	CV2	CV3
行数	9	32	30	8	23	23	22	25
TIF の数	1	1	1	2	3	1	2	1

表 2: 実行ステップ数の比較 (実験 1)

プログラム		quicksort	heapsort	sim	nsieve	LKF 15	CV1	CV2	CV3
ステップ数 [step]	LS†	13.7	7.3	21.0	13.0	48.8	9.3	21.3	17.3
	CVCS-2	10.7	6.0	18.0	10.5	43.8	7.3	14.7	13.3
速度向上率‡[%]		28.0	21.7	16.7	23.8	11.4	27.4	44.9	30.0

† LS:List Scheduling

‡ 速度向上率 [%] = (LS–CVCS-2)/CVCS-2 × 100

表 3: 実行ステップ数の比較 (実験 1) —ALU2 並列—

プログラム		quicksort	heapsort	sim	nsieve	LKF 15	CV1	CV2	CV3
ステップ数 [step]	LS†	11.7	7.3	13.0	13.0	39.5	9.0	14.7	17.0
	CVCS-2	9.0	5.3	11.0	7.0	27.8	6.3	10.0	12.7
速度向上率‡[%]		30.0	37.7	18.1	85.7	42.1	47.3	47.0	33.9

表 4: 実行ステップ数の比較 (実験 2)

プログラム		LKF1	LKF2	LKF3	LKF4	LKF5
ステップ数 [step]	LS	16.7	19.1	8.3	14.6	11.9
	TS†	13.8	19.9	6.9	13.8	11.8
	CVCS-2	12.8	18.2	7.4	12.8	11.0
速度向上率 (LS:TS)* [%]		21.0	-4.0	20.3	5.8	0.8
速度向上率 (LS:CVCS-2)** [%]		30.5	4.9	12.2	14.1	8.2

† TS:Trace Scheduling

* 速度向上率 (LS:TS)[%] = (LS–TS)/TS × 100

** 速度向上率 (LS:CVCS-2)[%] = (LS–CVCS-2)/CVCS-2 × 100

5 おわりに

本稿では，我々が提案しているスケジューリング手法 CVCS のレジスタ割当を行わないバージョン CVCS-1 を拡張した CVCS-2 のアルゴリズムを述べた．また，CFG からアセンブラ命令列を生成する際に生じる JUMP 命令の最小化問題をグラフの最大マッチングを求める問題へ定式化した．

CVCS-2 の評価を行うために実行ステップ数に関する簡単な実験を行った．実験の結果，CVCS-2 はリストスケジューリングに比べ，11.4%～85.7% 実行ステップ数が減少した．さらに，実行ステップ数に関してトレース・スケジューリングと CVCS-2 との比較を行った結果，CVCS-2 はトレース・スケジューリングとほぼ同等の性能を得ることが確認できた．

今後，レジスタ割当を考慮に入れた CVCS-3 の開発を行う．レジスタに制限を加え，CVCS-3 と他の広域コードスケジューリング手法との比較を行う予定である．

参考文献

[1] 井上 昭彦, 赤星 博輝, 冨山 宏之, 若林 一敏, 安浦 寛人, “コンディションベクタを用いたコンパイラの最適化,” 情報処理学会研究報告, 95-ARC-110, pp.57–64, 1995 年 1 月.

[2] K.M. Chandy, and J.R. Dickson, “A Comparision of List Schedules for Parallel Processing Systems,” *Communications of the ACM*, vol.17, pp.685–690, Dec. 1974.

[3] K. Wakabayashi and T. Yoshimura, “A Resource Sharing and Control Synthesis Method for Conditional Branches,” *ICCAD-89*, pp.62–65, Nov 1989.

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

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