計算機アーキテクチャ 115－4
設 計 自 動 化 78－4
（1995. 12. 14）

レジスタ割り付けを同時に行う命令レベル並列プロセッサ向け広域コードスケジューリング手法

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

本稿ではレジスタ割り付けと広域コードスケジューリングを同時に行う手法を提案する．従来のレジスタ割り付け手法は命令レベル並列性を十分に考慮に入れておらず，プロセッサの性能を引き出すことは困難である．レジスタ割り付けを行うと同時にスケジューリングを行うことにより，以下の利点がある．1) レジスタ割り付けにおいて生じるデータ依存関係によりスケジューラが制約を受けない．2) レジスタ割り付けにおいて生じるスピルコードもスケジューリングの対象となる．ベンチマーク・プログラムを用いた実験により，本手法の有効性を評価した．

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，
レジスタ割り付け，コンパイラ

A Global Code Scheduling Technique with Register Allocation for Fine Grain Parallel Processors

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

We propose a scheduling algorithm which performs global code scheduling and register allocation simultaneously. In most of previous register allocation techniques, instruction level parallelism is not considered. So it is difficult to bring out the maximum performance of processors. Scheduling instructions simultaneously with register allocation, we have the advantages; 1) In the scheduling process, there are no constraints by extra data dependences which the register allocator produces. 2) Spill codes inserted by the register allocator itself can be also scheduled simultaneously. We have evaluated our scheduling algorithm using several benchmark programs.

Key Words : Condition Vector, Global Code Scheduling, Instruction Level Parallelism,
Register Allocation, Compiler

1 はじめに

これまで，レジスタ割り付けを同時に行う有効な広域コードスケジューリングは提案されていない．本稿では，レジスタ割り付けと広域コードスケジューリングを同時に行う手法を提案する．コンパイラにおいてレジスタ割り付けとスケジューリングは全く別のフェーズとして実行されてきた．従来のレジスタ割り付け法を命令レベル並列プロセッサに用いると，並列性を考慮に入れていないため，

- レジスタ割り付け前の段階では同時に実行可能な命令が割り付けにより同時に実行できなくなる
- 割り付けにより新たに生じるロード・ストア命令が他の演算と並列実行できる機会が少ない

という問題が生じる．従って，命令レベル並列プロセッサに対しては，レジスタ割り付けとスケジューリングは互いに協調して行う必要がある．提案手法はスケジューリング過程においてレジスタを同時に割り付けることにより，上記の問題に対する一つの解決策を与える．

2章においてレジスタ割り付けとスケジューリングの関係について述べる．3章において我々が従来提案しているスケジューリング手法について簡単に説明を行い，スケジューリング過程においてレジスタ割り付けを行う方法について4章で述べる．5章で実験を行い，6章でまとめる．

2 レジスタ割り付けとスケジューリングの関係

コンパイラは，全ての変数や定数にレジスタを割り付けることができない場合は，レジスタの値をメモリに退避する命令を生成しなければならない．この命令は一般的にスピルコードと呼ばれる．命令レベル並列性を持たないプロセッサにおいては，スピルコードにより実行サイクル数は確実に増加するため，「スピルコード生成を最小限に抑える」ことがレジスタ割り付けの目標である．しかし，命令レベル並列プロセッサにおいてはスピルコードの増加により実行サイクル数が増加するとは限らない．スピルコードが他の演算と並列に実行可能な場合があるからである．コンパイラは，実行サイクル数を増加させないレジスタ割り付けを行う必要がある．

コードスケジューリングを実行する時期は，レジスタ割り付けとの関係により2通り存在する．プリパス・スケジューリングとポストパス・スケジューリングである [1]．プリパス・スケジューリングはスケジューリング後にレジスタ割り付けを行う方法である．ポストパス・スケジューリングはレジスタ割り付けを行った後にスケジューリングを行う方法である．プリパス，ポストパス・スケジューリングの利点および欠点を表1にまとめる．本稿では，上記の2つのアプローチとは異なり，スケジューリングと同時にレジスタを割り付けるというアプローチをとる．このアプローチは，

- レジスタ割り付けにより生じる新たなデータ依存関係によりスケジューラが制約を受けない
- 割り付けにより生じるスピルコードを他の演算と同時にスケジューリングできる

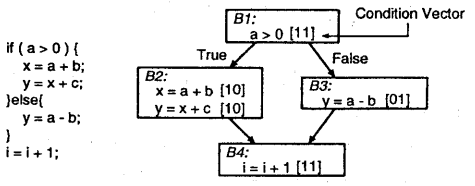
という利点を持つ．局所スケジューリングとレジスタ割り付けを同時に行う方法は提案されているが，広域コードスケジューリングは行わない [1]．提案する手法は，広域コードスケジューリングと同時にレジスタ割り付けを行うものである．

以降，我々が提案している広域コードスケジューリング手法を述べ，スケジューリング過程においてどのようにレジスタを割り付けていくかについて説明する．

3 コンディションベクタを用いたスケジューリング手法

[2] において我々は広域コードスケジューリング手法を提案している．本章では，[2] で提案したスケジューリング手法について説明し，次章において本稿で提案するレジスタ割り付けの手法について説明を行う．

3.1 コンディションベクタ



(a) プログラム　(b) (a)に対するコントロールフロー・グラフ

図 1: コンディションベクタ

コンディションベクタ (CV:Condition Vector)[3] はスケジューリング対象となる各演算に与えられ

表 1: プリパスおよびポストパス・スケジューリングの利点，欠点

	利点	欠点
Prepass Scheduling	レジスタ割り付けの結果生じる新たなデータ依存関係をスケジューラが考慮する必要がないため，スケジューリング時の制約が少ない．	レジスタの生存区間が伸び，その結果生じるスピルコードが増加する．スピルコードは他の演算と並列に実行できない．
Postpass Scheduling	スピルコードをスケジューリングの対象とすることが可能となる	レジスタ割り付けの結果生じる新たな依存関係により，スケジューリング時の制約が多くなる

るビットベクトルであり，演算の実行条件を表現する．ベクトルの各要素はプログラム中の分岐命令の成立，不成立を表している．図 1(b) は (a) のプログラムに対するコントロールフロー・グラフを示しており，各演算に与えられたビットベクトルが CV である．図 1において，CV[10] をもつ演算は条件 $a > 0$ が成立した時のみ実行される．また，CV[11] を持つ演算は条件に関わらず実行される．

3.2 スケジューリング手法

CVCS(Condition Vector Code Scheduling) は CV を用いた広域スケジューリング手法である．スケジューリングの対象となる範囲 (リージョンと呼ぶ) は 1 つ又は複数の連続した条件分岐文からなるプログラムである．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リージョンとする．スケジューリングは以下の手順により行う．

1. リージョン内のデータフロー・グラフ (DFG) を作成する．DFG の各節点は演算であり，枝はデータフロー関係を表現する．リージョン全体で 1 つの DFG を作成する．
2. リージョンに含まれる全ての演算に CV を与える．
3. リージョン内でもっとも優先度の高い演算を選択する．ある演算の優先度はその演算に後続する演算よりも大きいものとする．演算 B が演算 A に後続するとは，演算 B は必ず演算 A の終了後に実行されることを意味する．
4. 3において選択した演算を可能な限り早い時期に実行されるようにタイムテーブルに割り当てる．タイムテーブルとはプログラム中の各演算が実行されるマシンサイクルを表現したものである．各マシンサイクルで使用可能な演算器の数は決まっているため，スケジューラは各サイクルにおいて使用可能な演算器以上の演算器を使用しないように管理しなければならない．CVCS では CV を用いることにより演算器の管理を行う．

図 2に図 1(a) のプログラムの CVCS によるスケジューリング過程を示す．図 2において $x = a + b$ は条件 $a > 0$ が判定する前に実行される．このように，条件分岐の評価結果を待たずに，演算を開始することを投機的実行という．投機的実行を行うと演算の CV は 1 の要素が増えるように変化する．同図において，$i = i + 1$ は CV が [10][01] に分解され，それぞれ可能な限り早いサイクルにスケジューリングされる．$i = i + 1$ が分割され，プログラムのより上流へ移動したことを示す．これは投機的な実行ではない．従来，投機的な演算の移動と投機的ではない移動は区別して行われる．CVCS では投機的な演算の移動と投機的ではない演算の移動を同一のデータ構造を用いて表現することが可能である．スケジューリング時に演算の移動の区別を行わないことは，より高速なコンパイルを可能とする．

4 レジスタ割り付け法

本章では CVCS を拡張し，スケジューリングとレジスタ割り付けを同時に行う手法を提案する．本手法では，レジスタ割り付けも CV を用いて行う．スケジューリングとレジスタ割り付けに用いられるデータ構造を統一することにより，より高速なコンパイルを実現する．

4.1 レジスタが必要な箇所

レジスタは DFG の枝がタイムテーブルのサイクルを横切るところに必要となる (図 3)．サイクル t においてレジスタが必要となる場合は以下の 3 通りに分けられる．

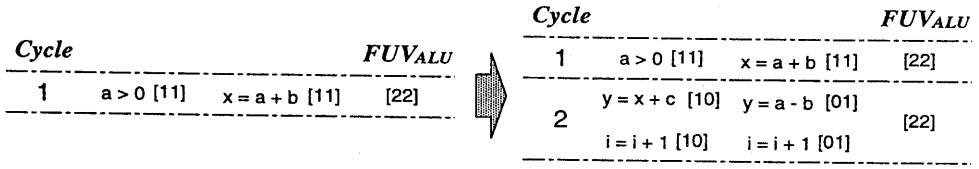
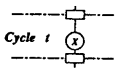
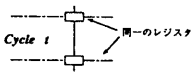


図 2: スケジューリング過程

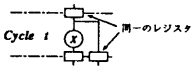
- tにおいて新たに値が定義された場合 (図 3(a))
- tでは値が使用されずにt以降に使用される場合 (図 3(b))
- tおよびt以降において使用される場合 (図 3(c))



(a) tにおいて新たに値が定義される



(b) tでは値が使用されずにt以降に使用される



(c) tおよびt以降において使用される

図 3: レジスタが必要となる場合

図 3(a) の割り付けの場合，2 つのレジスタには同一のレジスタを割り付けても，異なるレジスタを割り付けても良い．図 3(b) のようにレジスタの値があるサイクルでは使用されずにそれ以降で使用される場合，図の 2 つのレジスタには同一のレジスタを割り付けなければならない．図 3(c) の場合，tで使用されかつt以降にも使用される値には同一のレジスタを割り付けなければならない．新たに定義される値を格納するレジスタには余っているレジスタを割り付ける．

4.2 レジスタの CV による管理

一般的にプロセッサにはレジスタ数に制限がある．スケジューラは各サイクルにおいて必要とするレジスタ数が使用可能なレジスタ数を越えないようにしなければならない．CVCS はレジスタの管理を CV を用いて行う．演算と同様にレジスタにも CV を与え，各サイクルにおいて使用されるレジスタの CV の和 ($FUVREG$) を求める．その和ベクタの要素の最大値が使用可能なレジスタ数を越えないようにスケジューリングを行う．演算Oが定義するレジスタに与える ($RCV(O)$ とする) は以下の式により求める．

$$RCV(O) = CV(O_1) \ or \ CV(O_2) \cdots$$
$$O_1, O_2, \ldots \in SUCC(O)$$

ここで orは論理和，$CV(O)$ は演算Oの CV，$SUCC(O)$ は演算Oに後続する演算の集合である．演算Oのスケジューリングの時点では演算$O_i \in SUCC(O)$ はまだスケジューリングされていない．O_iはスケジューリングされる箇所によりその CV が変化する可能性があり，Oがスケジューリングされる時点でのO_iの CV は判明していない．演算 Oが定義するレジスタに割り当てる CV を求める際は，$SUCC(O)$ に含まれる全ての演算がOが割り当てられている次のサイクルにスケジューリングされるものと仮定する．

4.3 スピルコードの挿入

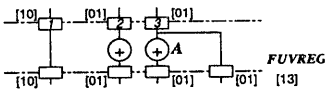
ある演算をスケジューリングする際，レジスタが不足して割り当てられない場合，以下の 2 通りの対処法が存在する．

1. 別のサイクルへのスケジューリングを試みる．
2. Load/Store 命令 (スピルコード) を挿入する．

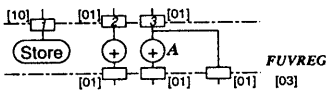
1の方法を選択した結果，スケジューリングを行っている演算がクリティカルパス上の演算であれば，クリティカルパスが伸び実行サイクル数が増加する．そのため，我々はレジスタが不足した場合 2 の方法をとることにした．2の方法を選択した場合，どの値をスピルさせるかが問題となる．すぐに使用される値をストアした場合，その値はすぐにレジスタへロードしなければならず，多くのスピルコードが挿入される可能性がある．挿入されるスピルコードを可能な限り抑えるため，レジスタが不足した場合は最も遅く使用されると思われ

る値をストアする．以下にストアする値を決定する方法を示す．

CVCSでは異なるパスに存在する演算を同一サイクルにスケジューリングする．ストアする値を選択する際，異なるパス上で使用される値をストアしても意味がない．図4に例を挙げる．各サイクルにおいて使用可能なレジスタが2個であると仮定している．演算Aのスケジューリングによりレジスタが不足した場合，異なるパスで使用される値(1番のレジスタが格納している値)をストアしても$FUVREG$の最大値は変化しない(図4(b))．この場合3番のレジスタが格納している値をストアするべきである．



(a) 演算Aのスケジューリングによるレジスタの不足



(b) 無意味なスピルコード

図 4: 無意味なスピルコードの挿入

ある演算Oの割り当ての結果レジスタが不足した場合，以下に定義する集合の中のいずれかのレジスタが保持している値をストアさせる．

$Spill_{candidate}$ =
{reg | $RCV(reg)$ *and* $RCV(reg_O) = RCV(reg)$,
regはOがスケジューリングされているサイクル以降にも使用される}

ここで，*and* は論理積を表し，

- reg_O : 演算Oが定義する値を格納するレジスタ
- $RCV(r)$: レジスタrのCV.

である．$Spill_{candidate}$内のレジスタが格納している値はOと同一パスで使用される．

4.4 スピルする値の選択

$Spill_{candidate}$内のレジスタが格納する値のうち，ストア命令によりメモリへ格納する値を決定する．優先度の低い演算は遅くスケジューリングされるため，優先度の低い演算が使用する値をストアする．ストアする値を決定する手順を以下に示す．

1. $Spill_{candidate}$内の各レジスタにおいて，そのレジスタを使用する演算のうち最も優先度の高い演算を選択する．
2. 1で選択した演算のうち最も優先度の低い演算が使用する値をストアする．

つまり，

- $USE(reg)$: レジスタregを使用する演算の集合．
- $MaxPriority(S)$: 演算の集合Sのうちもっとも優先度の高い演算．

とすると，$reg \in Spill_{candidate}$のうち

$$MaxPriority(USE(reg))$$

を最小にするregが格納している値をストアする．

5 実験

レジスタ割り付けを同時に行うCVCSについて評価を行うために実験を行う．実験の手順は以下に従う．

1. gcc Ver2.6によりプログラムをコンパイルし，アセンブラコードを出力する．
2. 出力したアセンブラコードからリージョンを選択する．
3. リージョンに対してスケジューリングを行う．

ターゲットとして，ALUユニット2個，分岐ユニット1個，ロード/ストアユニット1個を持つ並列度が2のRISCプロセッサを仮定しており，各演算は1サイクルで実行を終了するものとする．また，ターゲットプロセッサは汎用レジスタを32本装備しているものとする．ベンチマークプログラムとしてUNIXのアプリケーションプログラム(*wc*, *sum*, *head*)を用いる．表2, 3, 4は各スケジューリング手法のリストスケジューリングに対する速度向上率，および各リージョンにおいて必要としたレジスタ数を示している．PCVCS(Postpass CVCS)はレジスタ割り付けを行わないポストパス・スケジューリングであり，CVCSR(CVCS with Resister allocation)はレジスタ割り付けをスケジューリン

表 2: プログラム wc(word count) における結果

	Speed Up	Register
List Scheduling	1.00(10.0)	7
Trace Scheduling	1.09(9.2)	7
PCVCS	1.09(9.2)	7
CVCSR	1.25(8.0)	5

( ) 内は平均サイクル数

グと同時に行う．CVCSR 以外の手法は全てポストパス・スケジューリングであり，レジスタ割り付けには gcc Ver2.6 が行ったものをそのまま用いている．速度向上率 *Speed Up* は以下の式により求める．

$$Speed\ Up = \frac{Cycle(ListScheduling)}{Cycle(X)} \times d$$

$Cycle(X)$ はスケジューリング・アルゴリズム X を適用した際のリージョンの実行サイクル数の平均である．実行サイクル数の平均は各実行パスの実行確率による荷重平均である．d はリージョンの実行がプログラム全体の実行に占める割合である．

表より，PCVCS はトレース・スケジューリング [4] とほぼ同等の性能が得られることが分かる．CVCSR を適用することにより，PCVCS と比較し平均 1.23 倍，トレース・スケジューリングと比較し平均 1.21 倍の速度向上が確認できた．実験では，レジスタ数が十分に大きく割り付けによりスピルコードは発生しなかった．従って，割り付けの結果生じるデータ依存関係によりスケジューラが制約を受けなかったことが速度向上の原因であると考えられる．

実験では，いずれの手法においても必要とするレジスタ数はほぼ変化しなかった．gcc は十分な量のレジスタを用いて割り付けを行っているため，無駄なレジスタの使用が数カ所存在した．そのため，*wc*, *sum* において CVCSR が使用するレジスタ数が減少したと思われる．他のアプリケーションではレジスタ数が増加する場合も確認された．これは，命令の広域コード移動を行うため変数の生存期間が伸び，他の変数とレジスタを共有できる機会が減少したためと考えられる．

6 おわりに

レジスタ割り付けを同時に行う広域コードスケジューリング手法を提案した．レジスタ割り付けとスケジューリングを同時に行うことにより以下の利点がある．

表 3: プログラム sum における結果

	Speed Up	Register
List Scheduling	1.00(23.5)	7
Trace Scheduling	1.11(16.1)	7
PCVCS	1.09(16.3)	7
CVCSR	1.24(14.3)	6

( ) 内は平均サイクル数

表 4: プログラム head における結果

	Speed Up	Register
List Scheduling	1.00(32.0)	5
Trace Scheduling	1.23(26.0)	5
PCVCS	1.19(27.0)	5
CVCSR	1.68(19.0)	5

( ) 内は平均サイクル数

- レジスタ割り付けにより生じる新たなデータ依存関係によりスケジューラが制約を受けない．
- レジスタ割り付けにより生じるスピルコードを同時にスケジューリングすることが可能．

実験の結果は，我々のスケジューリング手法がポストパス・スケジューリングに対して有効であることを示している．今後，プリパス・スケジューリングに対する本手法の有効性を評価する必要がある．また，スケジューリングとレジスタ割り付けを協調させた他の手法 ([1, 5]) との比較も行う予定である．

参考文献

[1] J.R. Goodman and W.C. Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks," *Proc. of Int'l. Conference Supercomputing*, pp.442–452, July 1988.

[2] 井上 昭彦, 赤星 博輝, 冨山 宏之, 若林 一敏, 安浦 寛人, "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," 情報処理学会研究報告, 95-HPC-57, pp.121–126, 1995 年 8 月.

[3] K. Wakabayashi and H. Tanaka, "Global Scheduling Independent of Control Dependencies Based on Condition Vectors," *Proc. of 29th Design Automation Conference*, pp.112–115, Nov 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] S.S. Pinter "Register Allocation with Instruction Scheduling," *Proc. of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp.248–257, June 1993.