高速に out-of-order 実行するためのレジスタ割当て手法

松崎秀則† 藤井寛子† 近藤伸宏†

効率的に out-of-order 実行を行うためには，コード中に逆依存や出力依存といった冗長な依存が存在しないことが望ましい．これらの依存が並列実行を阻害する恐れがあるからである．そこで本論文では，out-of-order 実行を効率的に行えるコードを生成するために，冗長な依存を発生させないレジスタ割当て手法を提案する．本手法はグラフ彩色ルーチンの一部を書き換えることで実現可能なシンプルな手法である．SPECint95 による評価では，ラウンドロビン方式よりもレジスタスピルを削減しつつ，発生する冗長な依存を大きく削減できることが確認できた．

A Register Allocation Method for an Out-of-order System

Hidenori Matsuzaki,† Hiroko Fujii† and Nobuhiro Kondo†

For the purpose of effective out-of-order execution, it is desirable that compiler doesn't introduce redundant dependences, like anti-dependence or output-dependence, into the target code. Because, these dependences could constrain the parallel execution. In this paper, we propose a register allocation method that doesn't introduce redundant dependences, for efficient out-of-order execution. This method is simple and can be implemented by rewriting a part of graph coloring method. And the experimental result shows that our method could reduce spilled registers and many of redundant dependences than round-robbin method.

1. 序　　論

近年プロセッサの処理速度を高速化するために，スーパスカラや VLIW と呼ばれる並列に動作する演算器を複数持った CPU アーキテクチャが主流になってきている．このようなアーキテクチャにおいては，コンパイル時の命令スケジューリング最適化によって，その実行パフォーマンスを大きく向上させることができる．スケジューリングによって命令間のレイテンシを隠すことが可能となるばかりでなく，命令間の並列性を引き出すことができるからである．

ところがコンパイラによって並列性が引き出されたコードが，命令実行時において必ずしも期待どおりのタイミングで実行されるとは限らない．コンパイル時には予測不能なキャッシュミス等が発生しうるからである．そこでこのようなダイナミックな要因によるペナルティを軽減するために，out-of-order 機構が提案され，広く用いられている．これはダイナミックにコードの順とは無関係に実行可能な命令を見つけ，キャッシュミス等の影響によって実行不可能となった命令より先に実行することで，そのペナルティを軽減するための方式である（out-of-order 機構にはコンパイラが並列性を引き出していないようなコードから並列性を引き出すという役割もある）．

out-of-order 実行において，ある命令が発行可能かどうかはその命令と依存関係にある先行命令が依存を満たすような形ですでに実行されているかどうかで判断される．この依存関係として逆依存や出力依存があるが，これらの依存はレジスタ割当てによって生じる冗長な依存関係であって本質的な依存関係ではない．そのためレジスタリネーミングと呼ばれる方式で回避することが可能である．そしてこのような依存の回避が out-of-order 方式における速度向上の鍵であり，多くの out-of-order 方式スーパスカラプロセッサにおいて採用されている．しかしこのハードによるレジスタリネーミング処理は実行時に複雑な制御を行うために CPU の周波数を上げられない原因となり，必ずしも最適な方法とはいえない．

逆に，もしレジスタ割当てによって冗長な依存関係が生じなければ，レジスタリネーミング機構のないシンプルなアーキテクチャでも効果的な out-of-order 実行が可能であるといえる．本研究ではそのような冗長な依存を発生させないレジスタ割当ての実現を目的と

† 株式会社東芝研究開発センター
Corporate Research & Development Center, Toshiba Corporation

している．しかしここで考慮すべき点が 2 つある．1 つ目は，レジスタ割当てとは無限個の疑似レジスタで表現されているコードを有限の物理レジスタにマッピングするという作業であり，必ずしも冗長な依存関係を生じないようにできるとは限らないということである．不可避的に発生してしまう依存関係については，out-of-order 実行に対して与える悪影響が小さいものであることが望ましい．2 つ目は，効率の良いコード生成のためには，少ないスピルでコードを実現する必要もあるということである．本研究ではこれら 2 つの問題も考慮に入れたレジスタ割当て手法を提案する．

以下 “冗長な依存関係” とは “プログラムの並列性を減少させる本質的ではない依存関係” であると定義したうえで，2 章で関連する研究について，3 章で本論文で提案するレジスタ割当て手法の詳細について，4 章で SPECint95 を用いた評価について，5 章で結論についてそれぞれ述べる．

2. 関 連 研 究

2.1 レジスタ割当て

レジスタ割当て手法としてはレジスタ干渉グラフに対するグラフ彩色が一般的である．レジスタ干渉グラフとは各レジスタをノードとし，レジスタの値が定義されている点が別のレジスタの生存区間内であれば，それらのレジスタに対応するノード間がエッジで結ばれているグラフである．生存区間が重なっている疑似レジスタには異なった実レジスタを割り当てる必要があるが，この問題をグラフにおいてエッジで結ばれたノードどうしに異なる色を塗ることに置き換えることによりこれを解決する．このとき使用可能な色の数が実レジスタ数に相当する．

このグラフ彩色のアルゴリズムとして最も一般的なものとして Chaitin のアルゴリズム[1] が知られている．さらにこれに対してスピルするレジスタの選択方法を改良した Briggs のアルゴリズム[2),3)]，レジスタ干渉グラフの生成/利用法を改良した Hierarchical Graph Coloring[4] 等が研究されている．このいずれの手法においても，レジスタ干渉グラフを生成し，そのグラフを実レジスタ数 N で彩色するという問題と置き換えることでレジスタの割当てを行っており，干渉グラフの作り方やスピルするレジスタの選択方法を変更することで Chaitin のアルゴリズムを改善している．

2.2 レジスタ割当てと並列度

疑似レジスタで表現されていたコードを有限な物理レジスタにマッピングする際に，コードが本来有していなかった逆依存や出力依存といった冗長な依存関係が生じることがあり，このような依存がコードの並列性を損なわせる．

そこでコンパイル中のどの段階でレジスタ割当てを行うのかが ILP を高めるうえで重要な要素となっている．一般にレジスタ割当てを先に行うとレジスタプレッシャが小さくてすむ代わりに命令間の並列度が損なわれ，一方でスケジューリングを先に行うと命令間の並列度をより引き出すことができる代わりにレジスタプレッシャが増大するという相反する性質がある．そこで目的に応じた最適化を行うために，レジスタ割当てと命令スケジューリングの関係についての様々な手法が研究されている．

Chang らは，まず命令スケジューリングを行い，その後でレジスタ割当てを行う方式を提案している[6]．疑似レジスタで表現されたコードをスケジューリングするため，スケジューリング時に最大限の ILP を得ることができる．また Norris らは，レジスタ割当てを行った後でスケジューリングを行うための Scheduler-Sensitive Global Register Allocator を提案している[7]．この方式では並列度を損なわないようなレジスタ割当てを行うことによって，レジスタプレッシャの増大を回避しつつ，スケジューリング時の並列性が低下するのを回避している．ただしこの方式は依存関係にない命令を干渉グラフ中で隣接ノードとすることを基本としており，レジスタプレッシャが高い場合を苦手としている．また得られる並列性も大きい代わりに処理も複雑なものとなっている．

そこで Norris らはコストの小さい現実的な解決方法も提案している[5]．この手法は実装が容易で，かつ優先順位固定方式やラウンドロビン方式よりも並列性を向上させることができる手法である．ただしこの手法においては，レジスタ数が大きくなるに従って，ラウンドロビン方式の方が大きな並列度を引き出すことが分かっており，レジスタ数が少ないシステムに向いた手法であるといえる．

本研究でも既存のグラフ彩色手法への簡単な追加によって実現可能な，かつ優先順位固定方式やラウンドロビン方式よりも out-of-order 実行に適したレジスタ割当て手法を提案する．ここで out-of-order 実行に適しているコードとは，コード中にプログラムの本質ではない冗長な依存関係（逆依存・出力依存）が存在しない，もしくは存在したとしてもそれが out-of-order 実行に悪影響を及ぼさないコードである．さらにレジスタスピル数も優先順位固定方式に近付けている．

3. アルゴリズム

本章では，レジスタ割当てによって発生する新たな依存関係を既存の依存関係上に隠蔽できるような実レジスタについては積極的に再利用し，隠蔽が不可能であるような場合でもその依存関係が out-of-order 実行に与える悪影響を最小とするためのレジスタ割当て手法について説明をする．本手法はプレスケジューリング方式に適用するレジスタ割当て手法である．また後述するように様々なレジスタ割当てアルゴリズムに対して適用可能であるが，本論文においては広く一般的に利用されている Chaitin のグラフ彩色手法をもとに説明を行う．

3.1 グラフ彩色手法と本手法の関係

グラフ彩色手法によってレジスタ割当てを行うために，まず最初にレジスタ干渉グラフを生成する．レジスタ干渉グラフにおいてノードは各レジスタであり，レジスタの値が定義されている点が別のレジスタの生存区間内であれば，それらのレジスタに対応するノード間がエッジで結ばれる．

つまり同時に生存しているレジスタどうしはエッジで結ばれていることになるので，エッジで結ばれたレジスタには異なった実レジスタを割り当てる必要がある．

レジスタ干渉グラフ G を k 個の色で塗り分けるグラフ彩色は以下のようにして行われる．

(1) レジスタ干渉グラフ G の生成.
(2) グラフ G の中で隣接ノード数が k 未満のノード n を見つけ，G からノード n とそこから出ているエッジをすべて取り除いたグラフ G' を作る．このとき取り除いたノード n をスタックに積んでいく.
(3) G が空になるか，隣接ノード数 k 未満のノードがなくなるまで G' を G と置き換えて (2) を繰り返す.
(4) すべてのノードが隣接ノード数 k 以上となった場合は,（スピルコスト/隣接ノード数）が最小となるノードをスピルの候補として登録して G から取り除き，再び (2) の作業に戻る.
(5) G が空となったら，スタックからノードを取り出しながら干渉グラフを再構築．その際，隣接するノードとは異なる色を取り出したノードに塗る.
(6) スタックが空になるか，取り出したノードに塗る色がなくなるまで (5) を繰り返す.
(7) 取り出したノードに塗る色がない場合，そのノードをスピルさせて再び (1) から作業を始める.
(8) すべてのノードに色が塗り終れば彩色完了.

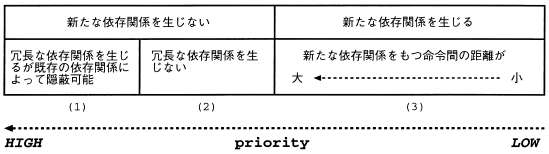


図 1 実レジスタの優先順序
Fig. 1 Priority order of hard register.

本手法はこれらの処理のうち (5) のフェーズにおいて疑似レジスタ（ノード）に割り当てる最適な実レジスタ（色）を複数の候補の中から選択するための手法である．レジスタ割当て手法として Chaitin のアルゴリズムをはじめ様々なアルゴリズム[2)~4)] が提案されているが，これらの手法は (1) や (4) のフェーズに対して工夫をしているものであるため，これらのアルゴリズムに対しても本手法は広く適用可能である．

3.2 実レジスタの選択

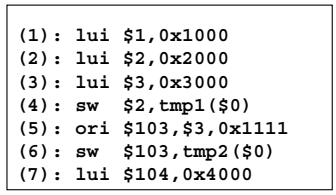
3.1 節のグラフ彩色手法の (5) のフェーズにおいて，ある疑似レジスタに割り当てることのできる実レジスタが複数存在した場合に，その中から割り当てるべき実レジスタを選択しなくてはいけない．通常，固定された優先順序を持たせて選び出す方法や，ラウンドロビンで選択する方法がとられている．一般に優先順位固定では効率良くレジスタの再利用をしてより少ない数の実レジスタでコードを実現するという点で優れているが，並列性は損なわれる．一方，ラウンドロビンでは再利用される実レジスタを空間的に分散させ，レジスタ割当てによって生じる冗長な依存関係をより少なくするという点において優れているが，レジスタスピル数は増大する傾向にある．

本手法ではそれらの候補に対して状況に応じて優先順序付けすることによって，out-of-order 実行に最適な実レジスタを割り当てる．具体的には，それを割り当てることによって冗長な依存関係を新たに発生してしまわないようなレジスタを最優先とし，もしそのようなレジスタが存在しない場合には，冗長な依存関係が out-of-order 実行に与える悪影響のなるべく小さくなるような実レジスタを優先させる．**図 1** に割り当てる実レジスタの優先順序を示す．以下ではこの図の (1)〜(3) にあてはまるレジスタを見つけ出すためのアルゴリズムについてそれぞれ説明する．

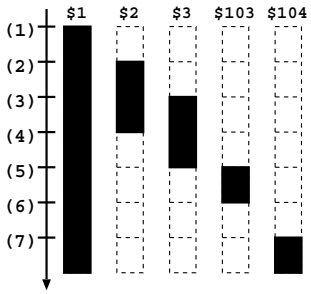
3.3 冗長な依存関係を生じない実レジスタ

3.3.1 すでに割り当てられた実レジスタ

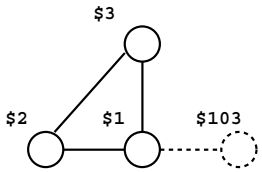
レジスタを有効に再利用するという観点から考える



(a)sample code



(b)live range



(c)interference graph

図 2 レジスタ割当てのサンプル例
Fig. 2 Sample of register allocation.

と，ある疑似レジスタに割り当てる実レジスタは，すでに他の疑似レジスタにも割り当てられているものであるのが望ましい．一方で不用意にそのような実レジスタを割り当てることは，逆依存や出力依存といった冗長な依存関係の発生を招く可能性がある．そこですでに他の疑似レジスタに割り当てられている実レジスタのうち，今割当てをしようとしている疑似レジスタに割り当てたとしても，冗長な依存関係を生じないような実レジスタを検索して，そのような実レジスタを優先的に割り当てる．

ここで新たな依存関係を生じないようなケースについて例をもとに説明をする．例として**図 2** (a) に示すような命令列へのレジスタ割当てを考えてみる．使用可能な実レジスタは$1〜$3，レジスタ$103，$104 は疑似レジスタであるとする．そしてここでは図 2 (a) に示すように一部の疑似レジスタにはすでに実レジスタが割り当てられているとする．この段階での各レジスタの生存範囲を図 2 (b) に，レジスタ干渉グラフを図 2 (c) に示す．ここで疑似レジスタ$103 に実レジスタを割り当てることを考えると，レジスタ干渉グラフから割当て可能な実レジスタの候補は$2 または $3 となる．**図 3** (a) にこの時点でのデータ依存グラフ，図 3 (b)，図 3 (c) には疑似レジスタ$103 に実レジスタ$2，$3 をそれぞれ割り当てた後のデータ依存グラフを示す．このように図 3 (b) においては新たに命令 (4)–(5) 間に逆依存が発生し，図 3 (c) においては新たに命令 (3)–(5) 間に出力依存が発生する．しかし図 3 (c) におけるデータ依存グラフは，レジスタを割り当てる前のデータ依存グラフと形状が変わらない．実レジスタ$3 を割り当てることによって命令 (3)–(5) 間に出力依存が生じるが，もともと命令 (3)–(5) 間にはフロー依存が存在していたためにそれが隠蔽されるためである．

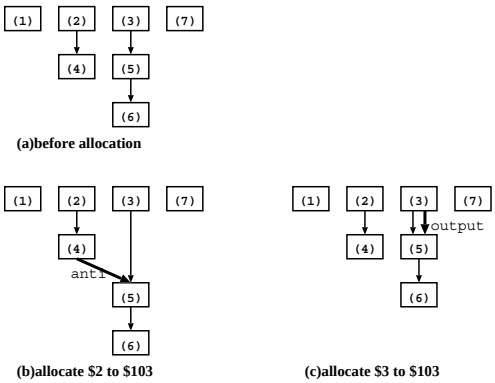


図 3 データ依存グラフ
Fig. 3 Data dependency graph.

このように新たに生じる依存関係がもともと存在していた依存関係と重なるように実レジスタを割り当てていけば，結果的に新たに冗長な依存関係が生じていないと見なすことができる．次にこのような実レジスタを発見するためのアルゴリズムを説明する．

ここでレジスタの定義とはレジスタに対する値の書き込みを，レジスタの使用とはレジスタからの値の読み出し意味する．この前提をもとに，疑似レジスタ r の生存区間 s（複数存在する可能性がある）ごとに以下の処理を行う．

(1) **先行依存命令から実レジスタ候補を検出**
まず生存区間 s 内の命令列の中から，疑似レジスタ r を定義している命令集合を見つける．そして，その命令集合が共通して依存している先行命令集合を検出し，この命令集合に含まれるすべての命令の使用レジスタを共通依存使用レジスタ集合 $c\_use_s(r)$ の要素とする．具体的には r を定義している命令が 1 つまたは複数存在し k は

その任意の要素を表すとすると，r を定義している命令 $def_s^k(r)$ それぞれがデータ依存グラフ中で後支配している命令集合 $pdom(def_s^k(r))$ を求め，最後にそれらの積集合を計算することで見つけることができる．

$$c\_pred_s(r) = \cap_{for\ all\ k} pdom(def_s^k(r))$$

そしてこの命令集合 $c\_pred_s(r)$ で使用されているレジスタを共通依存使用レジスタ集合 $c\_use_s(r)$ の要素とする．

このような実レジスタを疑似レジスタ r に割り当てたとしても，逆依存または出力依存が発生する命令間はすでに依存関係にあるため，この逆依存が新たに発生する冗長な依存とはならない．つまり，このようにして検出されたレジスタ集合 $c\_use_s(r)$ は冗長な依存関係を生じない可能性を持っている．

(2) **後続依存命令から実レジスタ候補を検出**

同様に生存区間内の命令列の中から，疑似レジスタ r を使用している命令集合を見つける．そして，その命令集合に共通して依存している後続命令集合を検出し，その定義レジスタをすべて共通依存定義レジスタ集合 $c\_def_s(r)$ とする．具体的には r を使用している命令が 1 つまたは複数存在し j はその任意の要素を表すとすると，r を使用している命令 $use_s^j(r)$ それぞれがデータ依存グラフ中で支配している命令集合 $dom(use_s^j(r))$ を求め，最後にそれらの積集合を計算することで見つけることができる．

$$c\_suc_s(r) = \cap_{for\ all\ j} dom(use_s^j(r))$$

そしてこの命令集合 $c\_suc_s(r)$ で使用されているレジスタを共通依存定義レジスタ集合 $c\_def_s(r)$ の要素とする．このようにして検出されたレジスタ集合 $c\_def_s(r)$ もまた冗長な依存関係を生じない可能性を持っている．

(3) **実レジスタ候補集合の計算**

(1)，(2) フェーズによって得たレジスタ集合の和集合 $ndep_s(r)$ を計算する．この集合が冗長な依存関係を生じない可能性を持った実レジスタ集合となる．

$$ndep_s(r) = c\_use_s(r) \cup c\_def_s(r)$$

ここまでで求めた実レジスタ集合 $ndep_s(r)$ は疑似レジスタ r を参照する命令に共通な先行/後続命令に関して冗長な依存関係を生じない実レジスタである．しかし実際にはこの中から実レジスタを割り当てたとしても，共通でない先行/後続命令との間に冗長な依存関係が生じる可能性は残っている．そこで次に (3) で求めた実レジスタ集合 $ndep_s(r)$ から，それを割り当てることによって新たな依存関係を生じる実レジスタを取り除いていく．

(4) **先行命令から実レジスタの絞り込み**

まず疑似レジスタを定義するすべての命令が共通には依存していない先行命令集合 $uc\_pred_s(r)$ を検出する．具体的には定義命令 $def_s^k(r)$ それぞれについて，コード並び順的に先行命令にあたる命令集合 $pred_s^k(r)$ を求め，それらの和集合を計算する．

$$all\_pred_s(r) = \cup_{for\ all\ k} pred_s^k(r)$$

そしてそこから $c\_pred_s(r)$ を取り除くことで $uc\_pred_s(r)$ が求められる．

$$uc\_pred_s(r) = all\_pred_s(r) - c\_pred_s(r)$$

そして最後に $uc\_pred_s(r)$ の命令の使用レジスタを非共通使用レジスタ集合 $uc\_use_s(r)$ として $ndep_s(r)$ から取り除く．そのような実レジスタは新たな依存関係を生じてしまうからである．

$$ndep_s(r) = ndep_s(r) - uc\_use_s(r)$$

(5) **後続命令から実レジスタの絞り込み**

同様に疑似レジスタを使用するすべての命令に共通には依存していない後続命令集合 $uc\_suc_s(r)$ を検出する．具体的には使用命令 $use_s^j(r)$ それぞれについて，コード並び順的に後続命令にあたる命令集合 $suc_s^j(r)$ を求め，それらの和集合を計算する．

$$all\_suc_s(r) = \cup_{for\ all\ j} suc_s^j(r)$$

そしてそこから $c\_suc_s(r)$ を取り除くことで $uc\_suc_s(r)$ が求められる．

$$uc\_suc_s(r) = all\_suc_s(r) - c\_suc_s(r)$$

そして最後に $uc\_suc_s(r)$ の命令の使用レジスタを非共通定義レジスタ集合 $uc\_def_s(r)$ として $ndep_s(r)$ から取り除く．そのような実レジスタは新たな依存関係を生じてしまうからである．

$$ndep_s(r) = ndep_s(r) - uc\_def_s(r)$$

ここまでの処理 (1)〜(5) を疑似レジスタのすべての生存区間 s に対して行い，最後にそれぞれの生存区間における集合 $ndep_s(r)$ の和集合を計算する．

$$NDEP(r) = \cup_{for\ all\ s} ndep_s(r)$$

このようにしてできた実レジスタ集合 $NDEP(r)$ は，すでに割り当てられた実レジスタのうち，冗長な依存関係を生じないレジスタの集合となる．この集合の中から疑似レジスタ r に割り当てる実レジスタを選択すれば，新たに冗長な依存関係を構築することなく実

レジスタを再利用できる．

また以上の過程で用いたデータ依存グラフはプレスケジューリング時にすでに生成している情報なので，小さなコストで実装可能である．

3.3.2 一度も割り当てられていない実レジスタ

3.3.1 項で割り当てるべき実レジスタを発見できなかった場合は，まだ一度も割り当てられていない実レジスタを割り当てる．ある疑似レジスタを割り当てる時点で，まだ一度も割り当てられていない実レジスタを割り当てても，他の命令との間に新たな依存関係を生じることがないからである．

3.4 冗長な依存関係を生じる実レジスタの優先順序付け

冗長な依存関係を生じないような実レジスタが存在しなかった場合，冗長な依存関係を生じるレジスタの中から，疑似レジスタに割り当てるべき実レジスタを選択しなくてはいけない．このような場合にも最適な実レジスタを選択することによって，それにより発生する新たな依存関係が out-of-order 実行に与える悪影響を最小限にとどめることができる．

本手法ではプレスケジューリング方式を採用している．この方式では命令スケジューリングが終了した後でレジスタ割当てを行うので，レジスタ割当ての段階ではすでに各命令がフェッチされる順序は確定している．つまり新たに発生する依存関係によって out-of-order 実行が妨げられることになる命令のフェッチされるタイミングが予測可能である．そこでそれらの命令間でフェッチされるタイミングが大きく異なるようにしてやることで，冗長な依存関係が out-of-order 実行に影響を与えない，もしくは与えたとしてもより小さいものに抑えることが可能となる．

次にこのようなレジスタを見つけるための処理の流れを示す．

(1) **全実レジスタの生存区間解析**

まず最初にその段階でのすべての実レジスタの生存区間を解析する．具体的には干渉グラフ中に存在するノード（すでに実レジスタが割り当てられた疑似レジスタ）すべてに対して，割り当てられた実レジスタの生存区間を解析する．

(2) **生存区間どうしの距離の計算**

次にすでに割り当てられた実レジスタの生存区間と実レジスタを割り当てようとしている疑似レジスタの生存区間との距離を計算する．ここで生存区間どうしの距離とは，一方の生存区間が終了してから他方の生存区間が始まるまでのサイクル数であり，あるレジスタに関して複数の生存区間が存在する場合にはそれらの区間すべてに関して距離を計算し最小値をその距離とする．

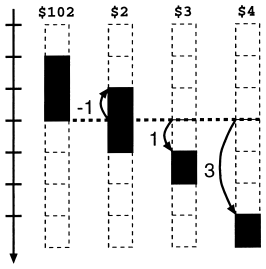


図 4 レジスタ生存区間の距離
Fig. 4 Distance of register live range.

図 4 のような生存区間を持つレジスタについて考えてみる．疑似レジスタ$102 に割り当てる実レジスタを選択しているとして，$102 と各実レジスタ$2，$3，$4 との距離はそれぞれ −1，1，3 となる．

(3) **生存区間どうしの距離の修正**

さらに計算された生存区間どうしの距離については修正を行う．計算された距離がある一定値 X 以上となった場合はその距離を一定値 X であるとする．生存区間が重なっている（干渉している）ものに関してはその距離は負数である．このようにしてすべての実レジスタに関して疑似レジスタの生存区間との距離が計算される．一般的に X の値としてはそれ以上距離が離れていれば依存の影響はなくなると考えられるような値を利用する．具体的にはターゲットとなるアーキテクチャのリオーダバッファサイズ等から算出することになる．こうすることにより X 以上の距離を持つ実レジスタは平等な優先順位を得ることとなるため，レジスタの再利用を促すことができる．一般的に X の値が大きければよりラウンドロビン方式の性質に近付き，逆に X の値が小さいほど優先順位固定方式の性質に近付くことになる．またターゲットとなるアーキテクチャを特定しないようなコードを生成したい場合には，X によるまるめを用いない方法も考えられる．

(4) **優先順序付け**

最後に (3) で計算された生存区間どうしの距離の大きい順に優先順序付けを行う．

以上の手順によって優先順位付けが完了したら，最も優先順位の高い実レジスタを疑似レジスタに対して割り当てる．

4. 性能評価

本論文で提案したレジスタ割当て手法によって out-of-order 実行に適したコード生成が可能となるかどうかについて評価を行う．out-of-order 実行に適しているコードとは “レジスタ割当てによって生じた冗長な依存の数が少なく” かつ “冗長な依存が生じたとしてもその距離が離れている” コードである．また “レジスタスピル数が少ない” ことも効率の良いコードの重要な要因である．そこで評価においては，本手法が従来の手法と比べてどの程度これらについて改善できているのかについて調査する．評価は本手法および比較対象となる従来手法によってコンパイルされたコードを，コンパイラ内部でスタティックに解析することにより行う．今回使用したコンパイラは，MIPS R4000 をターゲットとする GCC2.8.0 をベースにして，独自に基本ブロックレベルのプレスケジューリング機構，および比較対象となる各レジスタ割当て機構を組み込んだものである．ただしターゲットプロセッサがリオーダバッファサイズ 32 の out-of-order スーパスカラプロセッサであることを想定し，依存距離のまるめ値 X を 32 とする．評価用プログラムとして SPECint95 を用いた．

次に具体的な評価の方法を説明する．コンパイラは，まず最初に疑似レジスタで表現されたコードのまま基本ブロックレベルのプレスケジューリングを行う．スケジューリング方法は一般的なクリティカルパススケジューリングであり，スケジュール時にレジスタプレッシャ等は考慮にされていない．次にそのコードに対して Chaitin のアルゴリズムにをベースとした優先順位固定方式（以下，FF），ラウンドロビン方式（以下，RR），そして本提案方式（以下，ADV）の 3 通りの方式でレジスタ割当てを行う．なお優先順位固定方式においては，レジスタ番号昇順で割り当てる優先順位を固定している．そして，この過程において，(1) レジスタ割当てによって新たに生じた冗長な依存の数，(2) 新たに依存関係となった命令間の距離，(3) スピルの対象となった疑似レジスタ数，の 3 つについてスタティックな解析を行った．ここで (1)，(2) の “新たに生じた冗長な依存” としては，レジスタ割当て後の並列性に影響を与えるような依存のみをその対象とする．また使用可能な汎用レジスタ数が 32 個，および 64 個の場合についてそれぞれ評価することで，レジスタプレッシャが変化することによる影響を調査している．ただし SPECint95 において浮動小数点演算はわずかな数であり，プログラムの本質とは関係ないため，浮動小数点レジスタは 32 個固定とした．さらに X の値を変化させた場合のパフォーマンスの変化を調べるために $X = 8$ とした場合についての評価も行った．

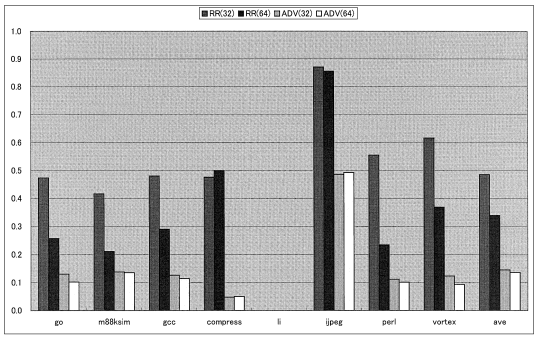


図 5 冗長な依存の数
Fig. 5 Number of redundant dependence.

4.1 冗長な依存の数

図 5 は，優先順位固定方式で発生した冗長な依存に対する，他の方式における冗長な依存の発生数の割合を示すグラフである．各ベンチマークプログラムごとに，左から RR（レジスタ数 32 個），RR（レジスタ数 64 個），ADV（レジスタ数 32 個），ADV（レジスタ数 64 個）の 4 本ずつの棒グラフで示す．また一番右側の ave は全プログラムの平均値を示している．棒グラフは，値が 1 であれば同一レジスタ数の FF と同じ数だけ冗長な依存が発生したことを，逆に値が 0 であれば冗長な依存はいっさい発生していないことを表している．

平均値を見てみると本方式を用いることによって優先順位固定方式に対して，レジスタ数 32 個で 85.4%，レジスタ数 64 個では 86.4%削減できているのが分かる．一方，ラウンドロビン方式ではレジスタ数 32 個で 51.4%，レジスタ数 64 個で 66.0%削減している．

この評価結果から次の 2 つのことが分かる．まず第 1 に冗長な依存を削減できるといわれているラウンドロビン方式に比べて，本方式の方が冗長な依存の発生数が大きく下回っているということである．レジスタ数 64 個の平均値で比べてみると，冗長な依存の発生数はラウンドロビン方式の約 40%となっている．この結果はラウンドロビン方式のような偶然性による冗長な依存発生の回避よりも，依存グラフを解析することによる明示的な手法がより有効であることを示している．

第 2 に，本方式がレジスタ数の変化にあまり影響されていないということが分かる．本方式においては，解析結果に応じて必要なレジスタ数が決定されるの

で，たとえばレジスタ数 32 個でその必要数を満たしている場合には，その数が 64 個に増えたとしても発生する冗長な依存の数は変化しない．逆に十分な実レジスタが存在しなかった場合には，依存が発生する状態でレジスタを再利用しなくてはいけないケースも発生する．go や vortex ではこのようなケースにあてはまる関数が多く存在し，これらのベンチマークにおいては，レジスタ数が 64 個に増加することで発生する冗長な依存の数は大きく減少している．ただしこのようなケースにあてはまらない関数においてはレジスタ数が増加したとしても依存の数に変化はないため，ラウンドロビン方式に比べると，レジスタ数による差は小さなものとなっている．一方，ラウンドロビン方式においては実レジスタ数が少ない場合には分散度も小さくなるため，実レジスタ数の減少に応じて冗長な依存数が増加してしまっている．ちなみに優先順位固定方式に関しては，レジスタ数が変化しても依存の発生数はほぼ同数であった．このことから，本方式がレジスタ数の少ないシステムにおいても効率的なコードを生成することができるといえる．

4.2 冗長な依存間の距離

次に**表 1** にレジスタ割当てによって新たに依存関係となった命令間の距離の平均値を示す．この表において距離が “1” であれば 2 つの命令が隣り合っている状態を表し，“2” であればその命令間に命令が 1 つスケジュールされている状態を表す．またプログラム li のいくつかの項目が “-” と表現されているが，これはその条件のときに距離を計測する対象となる冗長な依存が存在しないことを表している．さらに最下行の average は，各レジスタ割当て方式ごとの依存間距離の平均値である．ここで注意しなければいけないのは，それぞれ距離を測定する対象となる依存関係が同一ではないため，必ずしも各手法どうしの数値の比較が成り立つわけではないということである．しかし，依存関係となってしまった命令間の距離がどの程度になるのかについて，全体的な傾向を知ることはできる．

まず最初に平均値を見てみると，本方式を用いることによって優先順位固定方式に対して，レジスタ数 32 個で 1.73 倍，レジスタ数 64 個では 2.25 倍依存距離が長くなっているのが分かる．つまり 4.1 節の結果とあわせて考えると，新たに発生した冗長な依存関係を大幅に削減しつつ，もし発生した場合でもそのような依存関係となる命令間の距離を離すことに成功しているといえる．また 4.1 節において ijpeg ではレジスタ数が 32 個から 64 個に増加しても冗長な依存をあまり削減できていないという結果が出ていたが，その場合でも依存間の距離は 2.46 から 5.61 と倍以上に開いているのが分かる．

表 1 冗長な依存の距離
Table 1 Distance of redundant dependence.

プログラム	レジスタ数 32			レジスタ数 64		
	FF	RR	ADV	FF	RR	ADV
go	2.49	3.94	6.27	2.31	3.64	4.20
m88ksim	2.91	4.35	4.27	2.72	3.57	4.70
gcc	2.45	4.46	7.14	2.20	4.09	5.30
compress	2.67	5.00	6.00	2.70	5.80	6.00
li	1.90	-	-	1.86	-	-
ijpeg	1.76	2.71	2.46	2.10	5.30	5.61
perl	1.79	2.24	2.12	1.75	2.19	2.03
vortex	2.55	4.59	4.76	2.66	3.79	2.66
average	2.30	3.67	3.98	2.27	4.53	5.10

表 2 レジスタスピル数
Table 2 Number of spilled register.

プログラム	FF	RR	ADV
go	236	259	239
gcc	2097	2158	2120
ijpeg	188	292	248
perl	82	83	83
vortex	223	290	259
sum	2826	3082	2949

4.3 レジスタスピル

次にレジスタスピルの発生数という観点から従来の手法との比較を行う．**表 2** にレジスタ数 32 個のときに各プログラムでスピルしたレジスタの数を示す．最下行の sum はスピル数の合計である．ここで表に出ていない m88ksim，compress，li についてはいずれの割当て方式でもスピルは発生していない．またレジスタ数 64 個の場合は gcc 以外ではスピルが発生しないため，これも評価の対象から外した．

sum を見てみると，ラウンドロビン方式では優先順位固定方式と比べてスピルレジスタ数が 9.1%増加しているのに対して，本方式では 4.4%の増加に抑えられている．このことから本方式におけるレジスタの積極的な再利用がレジスタスピルを抑制するために効果的に機能していることが分かる．

4.4 依存距離のまるめ値 X

最後にまるめ値 X の影響について評価を行う．評価の方法としてはレジスタ数 32 個での，$X = 8$ としてコンパイルした場合の依存間の距離の平均値およびスピルレジスタ数の合計を調べ，$X = 32$ とした場合と比較をする．その結果，本評価におけるデフォルト値 $X = 32$ の場合と比べて，依存距離については約 7.2%の減少，スピル数については約 0.5%削減されることが確認された．3.4 節で述べたように，X の値が小さいほど優先順位固定方式の性質に近付いているの

が分かる．このように依存距離による優先順位決定時のまるめ値 X を変化させることにより，生成するコードの性質を調整することが可能であると考えられる．

5. 結　　論

本論文では out-of-order 実行に適したコードを生成するためのレジスタ割当て手法について述べた．本手法ではグラフ彩色における実レジスタを割り当てるフェーズにおいて，out-of-order 実行に適した実レジスタを優先的に割り当てつつ，実レジスタの再利用も積極的に行う．性能評価において，優先順位固定方式やラウンドロビン方式に比べて冗長な依存の発生を大きく削減しつつ，レジスタスピル数もラウンドロビン方式より小さく抑えられることが可能であるという結果が得られており，out-of-order 実行に適したコードが生成できることが確認された．

参 考 文 献

1) Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E. and Markstein, P.W.: Register allocation via coloring, *Computer Languages*, Vol.6, pp.47–57 (1981).
2) Briggs, P.: Register allocation via graph coloring, Ph.D. Thesis, Rice University (1992).
3) Briggs, P., Cooper, K.D., Kennedy, K. and Torczon, L.: Coloring heuristics for register allocation, *Proc. ACM SIGPLAN '89 Conference on Programming Language Design and Implementation* (1989).
4) Callahan, D. and Koblenz, B.: Register allocation via hierarchical graph coloring, *Proc. SIGPLAN '91*, pp.192–203 (1991).
5) Norris, C. and Fenwick Jr., J.B.: Understanding and Improving Register Assignment, *Proc. Euro-Par'99*, pp.1255–1259 (1999).
6) Chang, P.P., Lavery, D.M., Mahlke, S.A., Cheng, W.Y. and Hwu, W.W.-M.: The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors, *IEEE Trans. Comput.*, Vol.44, No.3, pp.353–370 (1995).
7) Norris, C. and Pollock, L.L.: A Scheduler-Sensitive Global Register Allocater, *Proc. Supercomputing '93* (1993).

(平成 12 年 8 月 30 日受付)
(平成 13 年 2 月 1 日採録)

松崎　秀則（正会員）

昭和 48 年生．平成 8 年早稲田大学理工学部電気工学科卒業．平成 10 年同大学大学院理工学研究科電気工学専攻修士課程修了．同年（株）東芝入社．現在,（株）東芝研究開発センターコンピュータ・ネットワークラボラトリー研究員．並列計算アーキテクチャ，および自動並列化コンパイラに関する研究に従事．

藤井　寛子

昭和 44 年生．平成 4 年慶応大学理工学部電気工学科卒業．平成 6 年同大学大学院理工学研究科計算機科学専攻修士課程修了．同年（株）東芝入社．現在,（株）東芝研究開発センターコンピュータ・ネットワークラボラトリー研究員．並列計算アーキテクチャ，および自動並列化コンパイラに関する研究に従事．日本ソフトウェア科学会会員．

近藤　伸宏

昭和 49 年生．平成 9 年早稲田大学理工学部情報学科卒業．平成 11 年同大学大学院理工学研究科修士課程修了．同年,（株）東芝入社．現在,（株）東芝研究開発センターコンピュータ・ネットワークラボラトリー研究員．命令レベル並列アーキテクチャ，コンパイラに関する研究に従事．