レジスタ生存グラフを用いたレジスタ割付けへのプロセッサ並列度の考慮

片岡正樹[†] 古関聰[††]
小松秀昭[††] 深澤良彰[†]

本稿では，命令レベル並列プロセッサを対象としたレジスタ割付けとコードスケジューリング技法を提案する．我々は，命令レベル並列性（ILP: Instruction Level Parallelism）を抽出する手法として，ガード付きプログラム依存グラフ（Guarded Program Dependence Graph）から生成されるレジスタ生存グラフを用いた手法を提案してきた．この手法は，レジスタアロケータとコードスケジューラの協調動作を考慮した手法の 1 つであり，これまでの手法では抽出するのが困難な，高い ILP を抽出できるという特徴を持っている．しかし，この手法は，プログラム中の変数の値を表す仮想レジスタどうしの生存区間の干渉を，プロセッサの並列度を考慮せずにレジスタ数以下に低減しているため，資源割付け時にレジスタ不足が起きる可能性があるという問題があった．このため，不必要な nop 命令や，レジスタの値をメモリに退避するスピルコードが挿入されてしまい，性能が低下してしまうことがあった．そこで，本手法ではプレスケジューリングを導入することにより，さらなる ILP の抽出を図るとともに，コード割付け時のレジスタ不足問題を解決する．SPECint2000 の中から，特徴的なループを持つプログラムの最内ループに対して本手法を適用したところ，実行サイクル数において従来手法より平均 8%の性能向上が見られた．

Considering Parallelism of Processor to Register Allocation Based on Register Existence Graph

Masaki Kataoka,[†] Akira Koseki,[††] Hideaki Komatsu[††]
and Yoshiaki Fukazawa[†]

We propose a register allocation and a code scheduling technique for instruction level parallel processors. We have suggested a method using Register Existence Graph generated from Guarded Program Dependence Graph in order to extract instruction level parallelism (ILP) in a program. This method is one of the methods considering both of the register allocation and the code scheduling, and can extract higher ILP compared to the previous methods. However, this method has a possibility of register shortage at the resource allocation phase because it reduces the interference among values in the live variables as if the processor has an infinite number of execution units, and therefore, unnecessary NOP instructions or spill instructions that move register values to and from memory are inserted at the resource allocation phase. In this paper, we introduce a method using pre-scheduling in order to extract higher ILP and avoid the above mentioned register shortage. The performance of this method is about 8% better than that of a previous method when applying it to some loop-intensive programs in the SPECint2000 benchmarks.

1. はじめに

現在主流となっている RISC プロセッサでは，レジスタアクセスのコストに比べて，メモリアクセスのコストが高い．そのため，メモリアクセスが頻繁に起こると，プログラムの実行効率は大きく低下してしまう．そこで，プログラムを高速実行するためには，コンパイラによるレジスタの利用に関する最適化が必須となってくる．本稿では，このメモリアクセスを最適化するための新しいレジスタ割付け手法を提案する．

メモリアクセスを減らすには，パイプラインが 1 つの場合には，レジスタの再利用性を高めて，メモリへのアクセスを最少化すればよい．しかし，パイプラインが複数ある場合には，レジスタ再利用性を高めすぎたことにより，命令を入れ替えることによって依存が発生しやすくなる．そのため，命令の入替えを十分

† 早稲田大学理工学部
School of Science and Engineering, Waseda University
†† 日本 IBM 株式会社東京基礎研究所
Tokyo Research Laboratory, IBM Japan, Ltd.

に行うことができず，プログラムの命令レベル並列性（ILP: Instruction Level Parallelism）を引き出すことが困難になる．そこで，十分な ILP を引き出すためには，スピルコードの最適化とともに，並列実行も考慮した手法である必要がある．

並列実行を考慮するためには，コードスケジューリングの動作を考慮しなければならない．コードスケジューリングとは，同時実行できる命令を見つけて ILP を抽出し，実行効率を向上させ，プログラムの高速実行を可能にする最適化である．これは，プログラム中で用いられる変数の値である仮想レジスタの同時刻に生存する数を結果的に増やすことになり，メモリに配置される仮想レジスタをなくそうとするレジスタ割付けとは逆の目的となっている．このレジスタ割付けとコードスケジューリングという 2 つの最適化は，どちらも不可欠な最適化であるため，いずれかを先に行うことになる．しかし，レジスタ割付けとコードスケジューリングのどちらを先に行ったとしても，お互いに依存しあってしまうという問題がある．そのため，十分な ILP を引き出すためには，互いの協調動作を考慮する手法である必要がある．

上記 2 つの条件を満たすためには，まず，命令の順序に依存せずに，一意となる中間表現が必要と考えられる．これは，命令の順序によって，抽出できる並列度が異なるのを避けるためである．次に，並列度を抽出するための命令入替え範囲を，レジスタが足りる範囲に限定して行う必要があると考えられる．これは，そのまま解こうとすると NP 完全問題であるレジスタ割付けとコードスケジューリングの協調技法を，ヒューリスティクスによって計算量を抑えつつ，最適解付近の解を求められるようにするためである．後者を実現するためには，すべての時刻において，あらかじめレジスタに同時に保持する変数の数を，CPU のレジスタ数以下にしておけばよい．しかし，CPU の並列度を考慮せずにこの操作を行うと，資源割付け時にレジスタが足りなくなることがある．たとえば，レジスタ数 2 で a，b という 2 つの値があり，$a+b$，$a-b$ という演算があるとする．ALU が 2 つの場合には，2 つの演算を同時に実行してレジスタ不足は起こらないが，ALU が 1 つの場合には，3 つのレジスタが必要になり，レジスタが不足する．

本稿では，このような問題を解決しつつ，高い ILP を抽出するために，レジスタ生存グラフを用いた手法を用いる．さらに，この手法に存在する，資源割付け時にレジスタが足りなくなる可能性がある問題点を，プレスケジューリングを導入することで，この可能性を低減するとともに，さらなる ILP の抽出を図る手法を提案する．

2. 関連研究

命令レベル並列プロセッサ向けレジスタ割付け手法としては，並列化レジスタ干渉グラフを用いたレジスタ彩色法[2),3)] があげられる．この手法は，並列性を保持するエッジを加えたレジスタ干渉グラフ[1)] に対して，彩色問題を解くことでレジスタ割付けを行っている．

しかし，エッジは中間コードの並び順に依存し，また並列実行に対して安全な見積りを行っているため，現実には干渉しないノード間にも，エッジが張られたままになることが多く，無駄なスピルコードが挿入されてしまうという問題があった．

一方，レジスタ割付けとコードスケジューリングを緩和した手法として，Integrated Code-Scheduling[6)~8)] があげられる．これらの手法は，大まかなコードスケジューリングを行ってから，レジスタ割付けを行い，再度コードスケジューリングをすることで，この問題を緩和させることに成功している．特に手法 6) の後続研究である手法 7) では，1 つの DAG でレジスタの使用状況と機能ユニットの使用状況を表し，この DAG のレジスタ・機能ユニットの空きに，部分 DAG を合成して埋めることで，on-the-fly な資源割付けを可能としている．

しかし，手法 6) では，レジスタ割付けの結果が中間コードの並び順に依存する問題は，解決されていない．さらに手法 7) にもいえるが，ILP を低い状態から高めていく手法であるため，つねに高い ILP を抽出できるとは限らない．また，手法 8) では，動的コンパイラを対象としているため，時間のかかる大域的なコードスケジューリングは行っていない．

このレジスタ割付けの結果が中間コードの並び順に依存する問題を解決し，大域的なコードスケジューリングを行う手法として，レジスタ生存グラフを用いたレジスタ割付け手法[4),5)] が提案されている．レジスタ生存グラフは，プログラム依存グラフ[9)]（以下 PDG）に実行条件を示すガード情報を付加した，ガード付きプログラム依存グラフ[4)]（以下 GPDG）から生成され，仮想レジスタをノードで，その値の生成と使用を有向エッジで表し，同時刻に生存している仮想レジスタを表すノードを等時刻線と呼ばれる線が横切るグラフである．このように，命令の依存関係から作成するため，中間コードの命令順序はレジスタ生存グラフには影響せず，並列化レジスタ干渉グラフでは切ることが困難であった，現実には干渉しないノード間のエッ

ジを切ることができるという特徴を持っている．

このレジスタ生存グラフを用いた手法では，このような特徴を用いて無駄なスピルコードが挿入されることを防いでいる．また，あるタイミングに同時に生存している仮想レジスタ数が，CPU の利用可能なレジスタ数（実レジスタ数）以下になるように，あらかじめ命令の実行順序に制約を加えておくことで，コードスケジューラとの統合アルゴリズムを実現している．

しかし，この制約によって保証されることは，各サイクルに存在する命令がすべて同時に実行された場合にレジスタが足りる，つまりレジスタ数制約を満たすということである．実際に同時実行できる命令数は ALU 数までであるため，レジスタ数制約を満たすための操作で ALU 数の考慮がされていないこの手法では，コード生成時にレジスタが足りなくなる恐れがあった．この場合には，コードスケジューラによる何もしない命令である nop 命令の挿入，もしくはレジスタの値をメモリに退避するスピルコードの挿入が行われてしまい，想定していた性能が出ないことがあるという欠点があった．

本稿では，この欠点を補うためにプレスケジューリングを行うことで，レジスタ不足を前もって検出し，これを回避するように時間的な依存を加えて，命令の入替え範囲を狭めておくことで，実際のコード生成時におけるレジスタ不足の可能性を低減する手法を提案する．

3. 本手法の概要

本手法は，大きく分けて 2 つのステージに分けることができる．1 つは最大干渉度低減操作を行うステージ，もう 1 つはプレスケジューリングとコード生成を行うステージである．それぞれについての説明を以下に記す．

3.1 最大干渉度低減操作

主に，レジスタ数制約を満たす操作を行う．その流れを**図 1** に示す．

プログラムの最内ループに対して，SSA 変換[10] を施し，ガード情報を付加して，GPDG を作成する．GPDG では，プログラム中の最内ループに存在する命令間の制御依存も，PDG のデータ依存と同様に表現できる．また，GPDG は最内ループの入口と出口を表す START ノードと END ノードを持ち，命令をノードで表現している．この GPDG の各ノードには，ガード情報が付加され，命令間のデータと制御の依存関係を有向エッジを用いて表現する．SSA 変換後のサンプルコードとそれから作成した GPDG を**図 2** に示す．

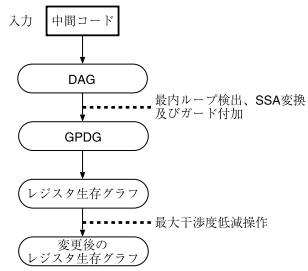

図 1 最大干渉度低減操作の流れ
Fig. 1 Flow of maximum interference degree reduction.

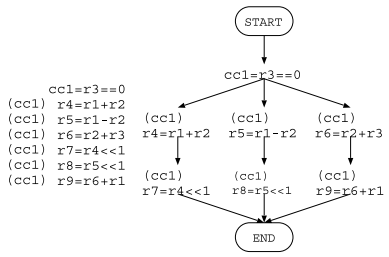

図 2 サンプルコードとその GPDG
Fig. 2 Example code and corresponding GPDG.

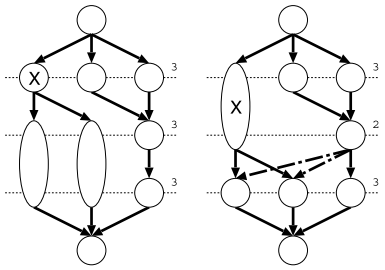

図 3 干渉度低減の例 1
Fig. 3 Example-1 of interference degree reduction.

次に，この GPDG からレジスタ生存グラフを生成する．このグラフの等時刻線で結ばれたノードの数は，その時刻における必要なレジスタ数を表し，干渉度と呼ぶ．その干渉度のすべての時刻における最大値を最大干渉度と呼ぶ．ある時刻での干渉度が，ターゲットとする CPU に備えられたレジスタ数を超えている場合，レジスタが不足することを意味するため，制限を加えることで最大干渉度を実レジスタ数以下に低減する必要がある．この最大干渉度を実レジスタ数以下に低減する操作，つまりレジスタ数制約を満たすための操作を，最大干渉度低減操作と呼ぶ．

干渉度を下げる方法は，大きく分けて 2 つある．1 つは，クリティカルパス長に影響が少ない演算の実行を遅らせる方法である（操作 1）．**図 3** のような例の場合，ノード X が表す仮想レジスタの値を使用する

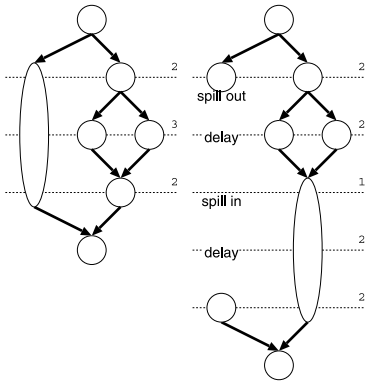

図 4 干渉度低減の例 2
Fig. 4 Example-2 of interference degree reduction.

演算を 1 サイクル遅らせて，干渉度を 1 つ下げることができる．

もう 1 つは，スピルコードを挿入することで，すぐには参照されなくてもかまわないデータをメモリに退避する方法である（操作 2）．**図 4** にその例をあげる．長い生存区間を持っていて，参照されない仮想レジスタが存在する場合などに，有効な方法であるといえる．

これらの操作によって，クリティカルパスが変わる可能性があるため，すべてのノードにおいて，自由度の再計算が必要である．ここでいう自由度とは，クリティカルパスに影響せず，どれほどそのノードの値を使用する演算を遅らせることが可能かという指標である．

最大干渉度低減操作のアルゴリズムを以下に記す．

(1) サイクルごとに干渉度を計算し，実レジスタ数を超えているサイクルを検出する．
(2) 干渉度を下げたいサイクルに対して，操作 1 を適用する．
(3) 操作 1 が適用できない場合，もしくは，操作後も干渉度が実レジスタ数を超えている場合には，以下の操作（操作 2）を行う．
  (a) 作業用リストに，レジスタ生存グラフ上で現在のサイクルのノードをすべて入れる．
  (b) 作業用リスト内のノードを自由度の低い順に並べ替える．
  (c) 実レジスタ数を超えない分，作業用リストの先頭からノードを取り除く．
  (d) 作業用リストの残りのノードすべてに対して，スピルコードを生成する．さらに，その値に対するスピルインノードを，2 サイクル後に付け加える．

まず，演算の実行順序を操作することによって，レジスタ数制約を満たせるか試す．それでも不足してしまう場合には，使用されるのが遅くてもかまわない仮想レジスタを，メモリに退避することでレジスタ数制約を満たす．スピルした値がレジスタに戻るタイミングは，現在の 2 サイクル後以降で，干渉度が実レジスタ数より少なくなる最初のサイクルとなる．

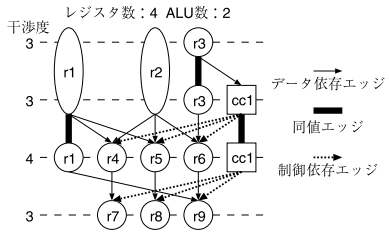

図 5 レジスタ生存グラフの例
Fig. 5 Example of Register Existence Graph.

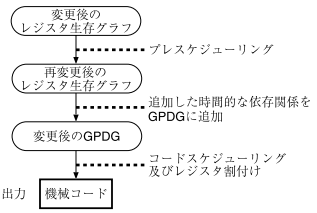

図 6 プレスケジューリングとコード生成の流れ
Fig. 6 Flow of pre-scheduling and code generation.

図 5 に，図 2 の GPDG から生成されたレジスタ生存グラフの例を示す．図中の楕円は，各サイクルでの干渉度を見やすくするため使用している．また，同値エッジとは，2 つのノード間で値が変わらないことを示し，後のサイクルでも使用される値を使用する場合，エッジの出所を見やすくするために使用する．今後は説明を簡略化するため，レジスタ数 4，ALU 数 2 というハードウェアを想定して説明する．また，図 5 のレジスタ生存グラフでは，すでにレジスタ数制約を満たしているため，最大干渉度低減操作を施す必要はない．

3.2 プレスケジューリングとコード生成

最大干渉度低減操作で加えた制約の補完と保証，最後に資源割付けを行う．この流れを**図 6** に示す．

変更後のレジスタ生存グラフに対して，ALU 数を考慮するため，仮想的な ALU への命令割付け操作を行う．これは，すべての時刻において，演算数が ALU 数以下であるか，もしくはその時刻に存在する優先順位の高い演算の中で，どのような演算の組合せでもレジスタが足りるかを確かめるために行う．

この操作中にレジスタ不足を検出した場合，バックトラックを用いて，優先的に発行した方がよい命令を

探索する．この探索方法を用いていくつか試行した結果，2 つの仮想レジスタを用いる演算の中で，参照が最後の仮想レジスタを 1 つ以上用いているものは，優先度の高い演算になり，定数との演算やシフト演算などは，優先度の低い演算になる傾向がある．ここで選択されなかったすべての命令に対して，新たに時間的な依存という制約を加え，その制約を満たすようにレジスタ生存グラフを変形する．このように，コード発行順序を限定して，レジスタが不足する可能性を低減している．もし，どの命令を選択したとしても，レジスタが不足してしまう場合には，スピルコードの挿入を行い，レジスタ生存グラフを変形する．この変形操作を，すべてのサイクルについて，レジスタ不足が起こらなくなるまで繰り返す．この最大干渉度低減操作とプレスケジューリングによって，レジスタ生存グラフに適用されたすべての制約を GPDG にも反映させ，レジスタ割付けとコードスケジューリングを行い，機械コードの生成を行う．

図 5 のレジスタ生存グラフは，コード発行時にレジスタが足りなくなる可能性がある例の 1 つである．この例では，2 サイクル目に r4，r5，r6 を生成する 3 つの演算が存在する．ALU 数が 2 であるため，3 つの演算を並列実行することができず，これらの中から 2 つ選ぶことになる．ここでは，この 3 つの演算の自由度は等しいので，リストスケジューリングでは発行の優先度が等しい命令となる．もし，優先度が異なる場合には，優先度の高いものから発行される．

演算の選び方としては，r4 と r5 を生成する演算組，r4 と r6 を生成する演算組，r5 と r6 を生成する演算組の 3 つが考えられる．しかし，もしコードスケジューラが r4 と r5 を生成する演算組を選んで発行した場合，演算後に r1，r2，r3，r4，r5 の 5 つの値を格納するためのレジスタが必要になるが，レジスタ数が 4 であるため，すべてはレジスタに乗り切らない．そのため，コードスケジューラによって ALU の 2 サイクル目で空きスロットが作られ，レジスタ上の値は**図 7** のレジスタ生存グラフに示すような振舞いをし，すべての演算を完了するために 5 サイクルかかることになる．ただし，他の演算組を選んだ場合には，レジスタが不足することはなく，すべての演算を完了するのに 4 サイクルで済むので，これが最悪時の性能ということになる．

このようなコード生成時における性能の低下を避けるために，r4 と r5 を生成する演算組が 2 サイクル目で選ばれないように，図 5 のレジスタ生存グラフを，**図 8** のように変形する．r4 と r5 を生成する演算組が

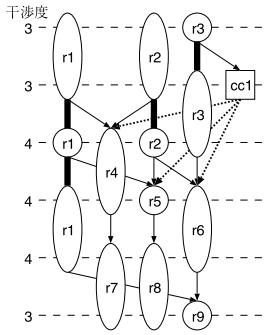

図 7 コード生成時における最悪時の性能
Fig. 7 The worst case at the time of code generation.

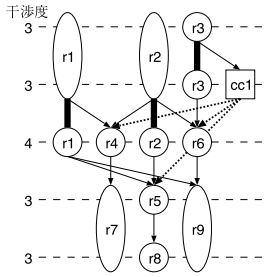

図 8 プレスケジューリング後のレジスタ生存グラフ
Fig. 8 Pre-scheduled Register Existence Graph.

選ばれないようにするには，r4 を生成する演算を遅らせる方法と，r5 を生成する演算を遅らせる方法の 2 種類が考えられる．この場合，どちらを選んでも性能は変わらないので，アルゴリズム上はどちらでも選べるようになっているが，ここでは説明の便宜上 r5 を生成する演算を遅らせることにする．

また，図 8 のレジスタ生存グラフの 3 サイクル目で 3 つの演算が存在しているが，自由度の最も低い r5 を生成する演算は，発行の優先度が高いため必ず選ばれ，r7 と r9 を生成する 2 つの演算の中から，1 つがコードスケジューラによって選択されることになる．この場合，どちらが選択されたとしても，演算後にレジスタが不足することはないため，このまま変形せずにしておく．これは，プレスケジューリングによって完全に発行順序を決めてしまうのではなく，後に動くコードスケジューラに，レジスタが不足しない程度に命令選択の余地を残しておくためである．

図 8 のプレスケジューリングを施したレジスタ生存グラフを基に，図 2 の GPDG に，さらに時間的な依存を付け加えて変更した GPDG を，**図 9** に示す．

r5 を生成する演算に対し，r4 と r6 を生成する各演算から時間的依存を示すエッジを加えて，r5 を生成する演算と，r4 と r6 を生成する各演算とが並列実行されないようにしている．この GPDG からレジスタ割

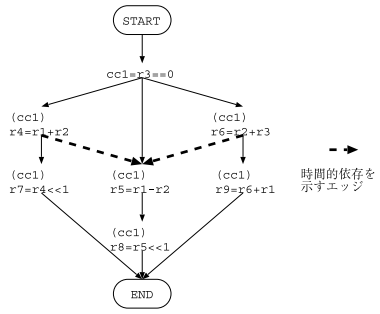

図 9 変更後の GPDG
Fig. 9 GPDG changed brought about by our method.

REG1	REG2	REG3	REG4
r1		r2	r3
r1		r2	r3
r1	r4	r2	r6
r1	r7	r5	r9
r1	r7	r8	r9

ALU1	ALU2
cc1=r3==0	
(cc1)r4=r1+r2	(cc1)r6=r2+r3
(cc1)r5=r1-r2	(cc1)r7=r4<<1
(cc1)r8=r5<<1	(cc1)r9=r6+r1

図 10 レジスタ割付けとコードスケジューリング結果
Fig. 10 Result of register allocation and code scheduling.

付けとコードスケジューリングを行った結果が**図 10** である．

4. アルゴリズム

本手法で用いている，プレスケジューリングと資源割付けの具体的なアルゴリズムを以下に記す．

4.1 プレスケジューリング

最大干渉度低減操作を施した後のレジスタ生存グラフを入力として，以下のようにプレスケジューリング操作を行う．

(1) 作業用リストに，現在のサイクルで発行可能な命令をすべて入れる．
(2) 作業用リスト内の命令を，自由度の低い順に並べ替える．
(3) 作業用リストから，与えられた ALU 数で考えられる演算の組合せの中で，自由度の合計が最低になるものをすべて作る．
(4) (3) で作成したすべての組合せに対して，命令実行後の干渉度を計算する．
(5) 干渉度が実レジスタ数を超えているものがない場合は，以下の操作を行う．
  (a) 組合せの中に入っているすべての命令を，作業用リストから除く．
  (b) 作業用リスト中の命令の自由度を 1 減らし，発行可能な命令がなくなるまで (1) へ戻る．
(6) すべての組合せにおいて，干渉度が実レジスタ数を超えている場合，以下の操作を順に試す．
  (a) 自由度の合計が最低にならないの演算の組合せを作り，(4) に戻る．
  (b) nop 命令や，利用されるまでの時間が最も長い値のスピルアウト命令との組合せを作り，(4) に戻る．スピルアウト命令挿入の場合には，2 サイクル後にスピルイン命令を挿入する．
  (c) 1 つも命令が発行できない場合，スピルアウト命令・スピルイン命令を挿入し，レジスタ生存グラフの再構築，命令の自由度の再計算をして，(1) に戻る．
(7) 干渉度が実レジスタ数を超えているものが 1 つ以上ある場合，以下の操作を行う．
  (a) 命令実行後の干渉度が実レジスタ数を超えている組合せが，すべて起きないようにレジスタ生存グラフを変形する．
  (b) 変形によって，生存区間が変化しなかった仮想レジスタを生成する命令を，作業用リストから除く．
  (c) レジスタ生存グラフの整合性がなくなるため，レジスタ生存グラフの再構築をする．
  (d) レジスタ生存グラフを再構築したことによって，命令の自由度も変わるので，自由度を再計算して，(1) へ戻る．

4.2 レジスタ割付けおよびコードスケジューリング

最大干渉度低減操作とプレスケジューリングによって，レジスタ生存グラフに加えられた制約を満たすように，GPDG に仮想的な依存を加え，レジスタ割付けおよびコードスケジューリングを行う．ここでは，あらかじめレジスタ数制約を満たすことで，レジスタアロケータとコードスケジューラの統合アルゴリズムを用いることを可能としている．

以下に本手法で用いている，レジスタ割付けおよびコードスケジューリングの統合アルゴリズムを記す．

(1) マシンサイクルを 1 とする．
(2) 作業用リストに GPDG 上での深さが 1 のノードをすべて入れる．
(3) GPDG のスタートノードにおいて，すでに生存している仮想レジスタを実レジスタに割り付ける．
(4) 以下の操作を，作業用リストが空になるまで繰り返す．
  (a) 作業用リストのノードを自由度の低い順に並べ替える．

(b) 作業用リストの先頭から，ALU 数を超えない数のノードに対応する命令を，現在のマシンサイクルの命令スロットに割り付け，それらを作業用リストから除く．
(c) 割り付けた命令が定義する仮想レジスタを実レジスタに割り付ける．
(d) 作業用リストにあるノードの自由度を 1 減らす．
(e) 命令スロットに割り付けられた命令の結果により，依存関係が満たされたノードをリストに加える．
(f) マシンサイクルを 1 増やす．

この結果から，機械コードを生成する．

4.3 計算量について

本手法で最も時間がかかると考えられる，プレスケジューリングの計算量について述べる．あるサイクルにおいて発行可能な命令数を n，ALU 数を r とすると，自由度の合計が最も低くなる命令の組合せは，${}_nC_r = n!/((n-r)! \times r!)$ 以下となる．すべての組合せにおいて干渉度が実レジスタ数を超えてしまう場合，nop 命令やスピルアウト命令と組み合わせるが，それぞれ ${}_nC_{r-1}, {}_nC_{r-2}, \cdots$ と計算回数は増える．

この計算量は，n が増えるにつれて，指数関数的に増えてしまうが，そのサイクルで発行可能な命令は非常に少なく，並列度を高める最適化を用いても $n \leq 10$ ほどであり，n は十分小さい．たとえば，$n = 10$，$r = 4$ としても ${}_nC_r = 210$ である．また，処理は各サイクルごとにバックトラックしているため，サイクルが深くなることによって計算量が増えることもない．

これらの理由から，本手法の計算量は実用的な範囲に収まっているといえる．

5. 実　　験

本アルゴリズムを用いた，レジスタアロケータおよびコードスケジューラの評価を行う．ハードウェアは，IA-64 に準拠したプロセッサとし，乗除算を除く整数演算命令と Store 命令には 1 クロック，Load 命令には 2 クロックかかるものとした．また，レジスタ割付けに対する定常的な実験をするため，キャッシュミスは起こらないものとした．このような条件下で，レジスタ数が 8，ALU 数が 2，4 の各場合について実験を行った．評価プログラムとして，SPECint2000 に含まれるプログラムのソースコードから，従来手法でコード発行時にレジスタが足りなくなる可能性があり，本手法を適用できる最内ループを抽出して用いた．使用したプログラムは，(1) gcc bc-optab.c，(2) voltex tree0.c，(3) twolf clean.c，(4) twolf findest.c となっており，**表 1**，**表 2** ではスペースの関係上，プログラム表記をここで示した番号表記とした．

表 1 レジスタ数 8 の場合
Table 1 Result of 8 registers.

	従来手法		本手法	
並列度	2	4	2	4
(1)	32	21	30	21
(2)	16	7	11	7
(3)	27	17	26	17
(4)	9	8	8	8

表 2 レジスタ数 32 の場合
Table 2 Result of 32 registers.

	従来手法		本手法	
並列度	2	4	2	4
(1)	28(14)	19(12)	28(12)	19(11)
(2)	11(10)	6(9)	11(9)	6(9)
(3)	26(11)	17(11)	26(11)	17(11)
(4)	8(9)	8(9)	8(8)	8(8)

評価値として用いたのは，GPDG のコードスケジューリング後の実行サイクル数である．実行サイクル数は実行時間に比例するため，これを比較することで，その性能を比較することができる．

従来手法として，文献 4) であげられている手法でクリティカルパス長を求め，我々の手法を適用したときの性能と比較した．ただし，従来手法ではクリティカルパス長が割付け方法によって変化するため，確率変数をクリティカルパス長とし，その割付け方法が起こる確率を求めて，期待値を評価値としている．レジスタ数が 8 の場合での結果を表 1 に，レジスタ数が 32 の場合での結果を表 2 に示す．ただし，レジスタ数が 32 の場合の括弧内の数字は，使用する最大レジスタ数を表す．

この実験により，本手法は並列度が低く，使用できるレジスタ数が少ない場合に，有効な手法であるといえる．これは，従来手法でレジスタが不足する際に挿入される nop 命令やスピル命令による遅延が，隠蔽しきれなかったためである．

また，並列度が高く，使用できるレジスタ数が少ない場合には，従来手法ではレジスタ不足が起きるのだが，nop 命令やスピル命令による遅延を，ALU の余剰スロットが隠蔽してしまうため，実行サイクル数で差が出なかった．

最後に，使用できるレジスタ数が多い場合には，並列度に関係なく，レジスタが不足する状況が起こりえなかったため，実行サイクル数は同じとなっている．しかし，最内ループ内で使用するレジスタ数を比較し

てみると，本手法の方が少なくなっている．これは，レジスタ生存グラフ上で，CPU の並列度の考慮がされていないために起こるためと考えられる．これにより，本手法はループアンローリングや Lazy Code Motion[11] といった，使用レジスタ数を増やす最適化と相性が良いと考えられる．たとえば，2，3 回ループアンローリングを行えば，レジスタ数 32 でもレジスタが足りなくなる．この場合には，レジスタ数 8 の場合と同じように，実行サイクル数に差が出ると思われる．

また，本手法の計算量と性能のトレードオフについてだが，本手法のオーダは指数関数並みと非常に大きい．しかし，探索範囲は非常に狭く，また実験により，並列度が低い場合には実行サイクル数の削減，並列度が高い場合には使用レジスタ数の削減と，実行するだけ価値がある性能差が現れていると思われる．

6. 終わりに

本稿では，命令レベル並列プロセッサ向けにコードを最適化するレジスタ割付けおよびコードスケジューリング技法を提案した．プレスケジューリングを用いることで，コード発行時にレジスタが不足する可能性の低減を図った．今後は，使用レジスタ数を増やす最適化手法と組み合わせることで，どれほど性能が向上するかを検証してみたいと考えている．

参考文献

1) Chaitin, G.J., Auslander, M.A., Chandro, A.K., Cocke, J., Hopkins, M.E. and Markstein, P.W.: Register Allocation via Coloring, *Computer Languages*, Vol.6, pp.47–57 (1981)
2) Norris, C. and Pollock, L.L.: A Scheduler-Sensitive Global Register Allocation, *Proc. ACM SIGPLAN '93 Conf. on Supercomputing*, pp.804–813 (1993)
3) Pinter, S.S.: Register Allocation with Instruction Scheduling: a New Approach, *Proc. ACM SIGPLAN '93 Conf. on Programming Languages Design and Implementation*, pp.248–257 (1993)
4) 小松秀昭，古関 聰，深澤良彰：命令レベル並列アーキテクチャのための大域的コードスケジューリング技法，情報処理学会論文誌，Vol.37, No.6, pp.1149–1161 (1996)
5) 古関 聰，小松秀昭，百瀬浩之，深澤良彰：命令レベル並列アーキテクチャのためのコードスケジューラおよびレジスタアロケータの協調技法，情報処理学会論文誌，Vol.38, No.3, pp.584–594 (1997)
6) Berson, D.A., Gupta, R. and Soffa, M.L.: Resource Spackling: A Framework for Integrating Register Allocation in Local and Global Schedulers, *IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pp.135–146 (1994)
7) Berson, D.A., Gupta, R. and Soffa, M.L.: GURRR: A Global Unified Resource Requirements Representation, *Proc. ACM Workshop on InterMediate Representations, Sigplan Notices*, Vol.30, pp.23–34 (1995)
8) Inagaki, T., Komatsu, H. and Nakatani, T.: Integrated prepass scheduling for a Java Just-In-Time compiler on the IA-64 architecture, *Proc. International Symposium on Code generation and optimization: feedback-directed and runtime optimization*, pp.159–168 (2003)
9) Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The Program Dependence Graph and Its Use in Optimization, *ACM Trans. Prog. Lang. Syst.*, Vol.9, No.3, pp.319–349 (1987)
10) Cytron, R., Ferrante, J., Rosen, B., Wegman, M. and Zadeck, K.: An Efficient Method of Computing Static Single Assignment Form, *Conf. Record of the 16th ACM Symposium on the Principles of Programming Languages*, pp.25–35 (1989)
11) Knoop, J., Rünthing, O. and Steffen, B.: Lazy Code Motion, *Proc. PLDI*, Vol.27, No.7, pp.224–234 (1992)

(平成 16 年 3 月 30 日受付)
(平成 16 年 8 月 25 日採録)

片岡 正樹

1980 年生．2004 年早稲田大学大学院理工学研究科情報科学専攻修了．同年同研究科コンピュータ・ネットワーク工学科博士課程進学，現在に至る．

古関 聰（正会員）

1969 年生．1998 年早稲田大学大学院理工学研究科電気工学専攻博士課程修了．同年日本 IBM（株）入社．以来，同社東京基礎研究所において，Java Just-in-Time コンパイラの開発に従事．工学博士．ACM 会員．

小松　秀昭（正会員）

1960 年生．1985 年早稲田大学大学院理工学研究科電気工学専攻修了．同年日本 IBM（株）東京基礎研究所入社．コンパイラ，アーキテクチャ，並列処理の研究に従事．博士（情報工学）．

深澤　良彰（正会員）

1953 年生．1976 年早稲田大学理工学部電気工学科卒業．1983 年同大学大学院博士課程修了．同年相模工業大学工学部情報工学科専任講師．1987 年早稲田大学理工学部助教授．1992 年同教授．工学博士．ソフトウェア工学，コンピュータアーキテクチャ等の研究に従事．電子情報通信学会，日本ソフトウェア科学会，IEEE，ACM 各会員．