シリーズパラレル型レジスタ生存グラフを用いたレジスタ割付け技法とその評価

近藤伸宏[†] 古関聰[††]
小松秀昭[††] 深澤良彰[†††]

プロセッサの性能を引き出すためには，レジスタ割付けやコードスケジューリングといった，プロセッサ資源の使用に関する最適化が不可欠である．命令レベル並列プロセッサの登場や，レジスタ数の増加などにより，これらの最適化の重要性は非常に高まっている．しかしながら一般に広く使われているレジスタ割付け手法では，プロセッサの並列性について十分に考慮されているとはいえない．本稿では，レジスタ生存グラフと呼ばれる，命令間の依存関係とコード中の並列性を構造に含むグラフを用いた手法を提案する．本手法では，このグラフをシリーズパラレル型のグラフに変形することによりコード中の並列性の抽出を行い，この結果と協調しながら同時にレジスタ割付けを行う．本手法を用いることにより，プロセッサの並列性を生かした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とが可能となる．

A Register Allocation Technique Using the Series-parallelized Register Existence Graph and Its Evaluation

Nobuhiro Kondo,[†] Akira Koseki,[††] Hideaki Komatsu[††]
and Yoshiaki Fukazawa[†††]

For the purpose of achieving very high performance of processors, register allocation and code scheduling are vital techniques because improving the usage of limited processor resources such as registers and ALUs vastly affects the performance. With the number of registers and ALUs increasing rapidly, the importance of these optimization techniques have become higher and higher. However, very few of existing register allocators takes instruction-level parallelism into account. This paper introduces a method using the graph that can express dependencies and parallelism among instructions simultaneously. Through making the graph series-parallel graph, code scheduling and register allocation are cooperatively conducted.

1. はじめに

プロセッサの動作速度の上昇にともない，レジスタやALUといったプロセッサ資源を有効に活用することが，計算機の性能の向上のために重要となってきている．従来より，さまざまなレジスタ割付け技法やコードスケジューリング技法が提案され，プロセッサ資源の有効活用が図られてきた．

プロセッサが演算を行う際に必要となるデータの格納場所としては，キャッシュ，メインメモリなどがあるが，最も高速にアクセスが可能なものにレジスタがある．また，プロセッサ動作周波数が高まるにつれ，キャッシュなどへのアクセスのコストがレジスタへのアクセスのコストに対し増大してきている．そのため，特に，演算に必要なデータをレジスタから取得し結果をレジスタに格納する，ロードストアアーキテクチャのプロセッサにおいては，演算に必要なデータをあらかじめレジスタに用意しておくということがプロセッサの持つ計算性能を引き出すことにつながる．どのデータをレジスタに置いておくのかを決定するのがレジスタ割付けである．

また，プロセッサが演算を行う順番は，演算に必要なデータの生産と使用や，分岐の方向などにより制約を受ける．これらの制約を満たしつつすべての命令の実行順序を決定するのがコードスケジューリングである．並列プロセッサ向けのコードスケジューリングを

† 株式会社東芝研究開発センター
Corporate Research and Development Center, Toshiba Corporation
†† 日本IBM株式会社東京基礎研究所
Tokyo Research Laboratory, IBM Japan. Ltd.
††† 早稲田大学理工学部
School of Science and Engineering, Waseda University

行う場合，これらの制約と，同時実行できる命令の制約を考慮しつつ，なるべく多くの命令を同時実行するようにスケジューリングを行う．これにより，プロセッサの性能を引き出すことができる．

本稿では，シリーズパラレルグラフを用いることにより，コード中に潜む並列性を引き出すようなレジスタ割付けおよびコードスケジューリングを行う技法を提案し，その評価を行う．

2. 本研究の背景

一般に広く用いられているレジスタ割付け技法に，グラフ彩色法のヒューリスティクスを用いたものがある[1)]．この手法では，レジスタをノードとし，仮想レジスタの生存区間の干渉を示すエッジでノードを結んだ，レジスタ干渉グラフと呼ばれるグラフを用いる．レジスタ干渉グラフ上でエッジで結ばれたノードどうしを互いに異なる色で塗るというグラフ彩色問題を，使用できる色数を実レジスタ数以下という制約の下で解くことによって，レジスタ割付けを完了させる．この手法は単純かつ強力であることから広く用いられるようになっている．

しかし，仮想レジスタの生存区間は，中間コードにおける命令の並び順によって変わってしまう．そのため，中間コードの性質が悪ければ，レジスタ割付けの結果も良くないものになってしまう可能性が高い．たとえば，中間コード生成の段階で，この手法の適用を前提とした並列性の抽出が行われていなければ，コード中の並列性をまったく考慮していないレジスタ割付けが行われてしまうことになる．これは，生成されたコードを並列プロセッサ上で実行させる場合には，そのプロセッサの性能を十分に生かしきれない可能性があることを意味している．

並列性の考慮に関しては，文献 2) や文献 3) で改良手法が提案されている．これらの手法は，ともにレジスタ干渉グラフに，コードの並列性を示すエッジを挿入することにより，並列性を確保しようとしている．このようなグラフを並列化レジスタ干渉グラフと呼ぶ．しかし，これらの手法は後に述べるような問題点が存在する．

コードスケジューリングにおいて，命令の実行順序を決定する制約の 1 つに，レジスタの使用に関するものがある．もし，レジスタが無限個存在すればこの制約は，データの生産と使用の関係に含まれるものとなる．しかし，現在のプロセッサではレジスタの数はせいぜい 32 個，次世代アーキテクチャと目されている IA-64 アーキテクチャでも整数用，浮動小数点数用それぞれ 128 個である．このようにレジスタ数が有限である場合，レジスタの使用に関して逆依存が発生し，レジスタの値を上書きするタイミングによっては，正しい計算結果を得ることができなくなってしまうことがある．このようなレジスタの使用に関する制約があることから，コードスケジューリングとレジスタ割付けの間には干渉が存在することになる．

コンパイラをコードスケジューリング，レジスタ割付けの順番で見ると，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うプリパス型，レジスタ割付けを行ってからコードスケジューリングを行うポストパス型がある．プリパス型のコンパイラでは，コードスケジューラが無限個のレジスタを仮定して ALU 資源を最大限に生かしたコードを生成し，そのコードに対しレジスタ割付けを行う．しかしながら，並列性を考慮しないレジスタ割付けを行うと，レジスタの使用に関する逆依存の発生により，並列性が損なわれてしまうことがある．また，レジスタプレッシャーの高い場合にはスピルコードを挿入する必要があるのだが，これによっても並列性が損なわれてしまうことがある．ポストパス型のコンパイラでは，レジスタ割付けにより中間コード中の仮想レジスタを有限個の実レジスタにマッピングしてから，コードスケジューリングを行う．この場合には，レジスタの使用に関する逆依存の発生により，スケジューリングの自由度が下がってしまい，ALU 資源を十分に生かしたコード生成が行えないことがある．このように，どちらの型においても，先に行った最適化の結果が次に続く最適化への制約となり，十分に最適化されたコード生成を行うことができないという問題がある．

このように，従来手法には，

(1) 中間コードにおけるコードの並び順という，コードを実行する際の本質的な制約とは無関係な制約を受ける，
(2) コードスケジューリングとレジスタ割付けの間にある干渉を考慮に入れていないために十分な最適化が行えない，

という問題点がある．

先ほど述べた，並列化レジスタ干渉グラフを用いた手法では，並列実行可能な命令が使用するレジスタの生存区間について安全な見積りを行っている．つまり，コードスケジューリングをするときに，並列実行するようにはスケジューリングされない命令であっても，並列実行できる可能性があるとしてしまうために，依存関係が複雑になってしまう．これらの手法で用いるレジスタの生存区間の見積り法は，中間コードにおけ

るコードの並び順から求めているため，(1) の問題点を解消できているとはいえない．

そこで，本稿では，シリーズパラレルグラフを用いることによって，上記の 2 つの問題点を解消した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する．本稿で提案する手法は，レジスタ生存グラフ[4]をシリーズパラレルグラフに変形することによりプログラムの並列性を生かしたコード生成を目指す．レジスタ生存グラフからは，コード中に含まれる命令間のデータ依存が得られる．しかし，ここで得られる依存関係は，従来のレジスタ干渉グラフから得られる依存関係と違い，レジスタの使用に関するものであるため，時間的にずれのある干渉を含んでいないという特徴がある．本手法では，レジスタ生存グラフをシリーズパラレルグラフに変形するが，それによりコード中の制約を満たしながら並列性の抽出を行うことが可能となる．本手法により，VLIW（Very Long Instruction Word）方式のプロセッサや，レジスタリネーミング機構やアウトオブオーダ実行機構を持たないプロセッサにおいて，レジスタの再利用による逆依存が原因となる性能低下を避けることができる．

3. 準　　備

ここで，本手法で使用するグラフ構造や，その他の用語について定義する．

3.1 SSA 変換

SSA（Static Single Assignment）変換[5]とは次のような変換である．まず，すべての変数に対して，その変数への代入が 1 度しか起きないようにする．そのために，複数の代入が起きるような変数については，リネーミングを行い，変数名を変える．次に，プログラムの意味が変わらないように制御の合流点に Φ ファンクションを入れる．

プログラムにこの変換を施すことにより，命令間の逆依存を解消することができる．

3.2 ガードつきレジスタ生存グラフ GPDG

GPDG(Guarded Program Dependence Graph)[6]とは，PDG[7]を拡張したものである．

GPDG はプログラム中の最内ループを表現するのに用いられる．GPDG のノードは最内ループ中の命令を表し，それぞれのノードには**ガード**と呼ばれる実行条件を付加する．また，GPDG は最内ループの始まりと終わりを示す START ノードと END ノードを持つ．

ガードと，命令が参照するデータの生産と使用に関する依存関係を，有向エッジを用いて表現する．これにより，GPDG 上では，制御依存がデータ依存に変換され，制御依存とデータ依存を統一的に扱うことができる．

3.3 レジスタ生存グラフ

レジスタ生存グラフは，GPDG から生成することができる．レジスタ生存グラフのノードはシンボリックレジスタをし，ノード間には，そのレジスタ値の使用を示す有向エッジが張られる．また，レジスタ生存グラフは最内ループの入口と出口を示す，TOP ノードと BOTTOM ノードを持つ．

レジスタ生存グラフでは，最内ループ内で定義されないレジスタ値の使用に関しては TOP ノードからエッジを張る．また，ループの出口で生存しているレジスタ値については，BOTTOM ノードに向けてエッジを張る．

また，レジスタ生存グラフには，仮想レジスタの生存時間の概念を導入することができる．

まず，レジスタ生存グラフの個々のノードに長さを持たせ，そのノードが示すレジスタ値がレジスタを占有している区間，すなわち，そのレジスタ値の生存時間を表すようにする．レジスタ値の生存時間の最小値は，そのレジスタ値を生産するのに要する時間であり，最大値は $+\infty$ である．

次に，レジスタ生存グラフにおいて**等時刻線**を定義する．等時刻線とは，レジスタ生存グラフにおいて，ある深さにあるすべてのパスを横切るように引いた線のことである．等時刻線とは，以下に示す条件を満たす．

(1) 等時刻線は，依存エッジと交わらない．
(2) 等時刻線は，他の等時刻線と交わらない．
(3) すべてのノードは，生存時間の最小値以上の本数の等時刻線と交わる．

この等時刻線は，最適化対象の先頭を基準に算出する．具体的には，TOP ノードを時刻の基準とし，その時刻を 0 とする．TOP ノードから BOTTOM ノードに向けて，等時刻線にたどりつくたびに時刻を 1 だけ増やす．

このようにして，レジスタ生存グラフへの時間の概念の導入が完了する．

上のような定義から，等時刻線上のノードは同時刻に生存していることになる．ある等時刻線上のノードの数をその時刻における**干渉度**と呼ぶ．そして，レジスタ生存グラフ中の干渉度の最大値を**最大干渉度**と呼ぶ．最大干渉度は，そのレジスタ生存グラフに対応した GPDG で表されるプログラムを，並列度 ∞ のプロセッサで実行させるようにコードスケジューリングを

行ったときに必要となるレジスタ数となる．この数を実レジスタ数以下にすることにより，対応する GPDG に対しどのようなコードスケジューリングを行ってもレジスタ割付けが可能となる．この数が実レジスタ数よりも大きかった場合にはそれを減らす操作が必要になるが，その操作を**最大干渉度低減操作**と呼ぶ．

3.4 シリーズパラレルグラフ

シリーズパラレルグラフ（Series-Parallel Graph）とは，グラフの 1 つのカテゴリで次のように定義される[8),9)]．

あるグラフを $G = (V, E)$ で表すことにする．ここで V はグラフ G の頂点の集合を，E はグラフ G のエッジの集合を表す．また，グラフ G の始点を $v_s(G)$，終点を $v_t(G)$ で表す．

(1) 始点と終点 2 点のみからなり，それらが単一のエッジで結ばれているようなグラフはシリーズパラレルグラフである．

(2) G_1 を $v_s(G_1)$ を始点，$v_t(G_1)$ を終点に持つシリーズパラレルグラフ，G_2 を $v_s(G_2)$ を始点，$v_t(G_2)$ を終点に持つシリーズパラレルグラフとする．このとき，

(a) グラフ G_1 の終点 $v_t(G_1)$ とグラフ G_2 の始点 $v_s(G_2)$ を重ね合わせて得られるグラフ G はシリーズパラレルグラフである．このシリーズパラレルグラフの始点は $v_s(G) = v_s(G_1)$ であり，終点は $v_t(G) = v_t(G_2)$ である．このような接続をシリーズ接続と呼ぶ．

(b) グラフ G_1 の始点 $v_s(G_1)$ とグラフ G_2 の始点 $v_s(G_2)$，グラフ G_1 の終点 $v_t(G_1)$ とグラフ G_2 の終点 $v_t(G_2)$ を重ね合わせて得られるグラフ G はシリーズパラレルグラフである．このシリーズパラレルグラフの始点は $v_s(G) = v_s(G_1) = v_s(G_2)$ であり，終点は $v_t(G) = v_t(G_1) = v_t(G_2)$ である．このような接続をパラレル接続と呼ぶ．

3.5 シリーズパラレル型レジスタ生存グラフ

レジスタ生存グラフのうち，シリーズパラレルグラフに属するものを**シリーズパラレル型レジスタ生存グラフ**と呼ぶ．本手法では，このシリーズパラレル型レジスタ生存グラフに注目して最適化を行う．

図 1 に示す GPDG に対応するシリーズパラレル型レジスタ生存グラフの例を**図 2** に示す．

この例では時刻 1 のときの干渉度が 6 になっているが，実際に通過しているノードの数は 8 である．これは，ノード j^* とノード h^{**} はそれぞれ 2 つあるが，元々同じレジスタであるため，二重に数えていないからである．このようなノードは，レジスタ生存グラフをシリーズパラレル型のグラフに変形するときにノードをコピーすることにより生成される．これらのレジスタは生存区間が重なっていれば，その時刻では同じレジスタを使用できる．

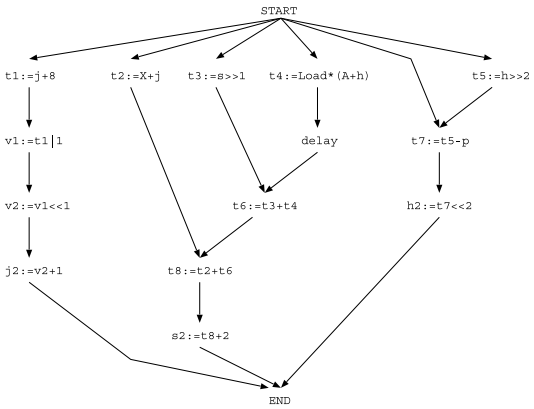

図 1 GPDG の例
Fig. 1 An example of GPDG.

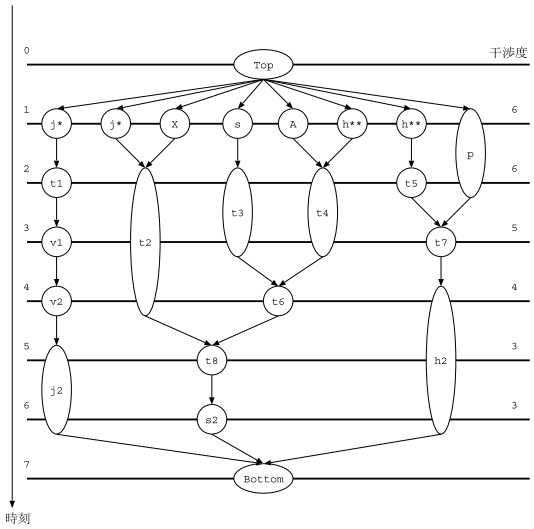

図 2 シリーズパラレル型レジスタ生存グラフ
Fig. 2 An example of series-parallelized register existence graph.

次に，いくつかの用語について定義する．図 2 の一部を取り出して情報を付加したのが**図 3** である．図 3 を用いて用語を定義する．

S スイート（Series Suite） シリーズパラレル型レジスタ生存グラフにおいて，シリーズ接続の連続で結ばれている，ノード，S スイート，P スイートをひとまとめにしたもの．

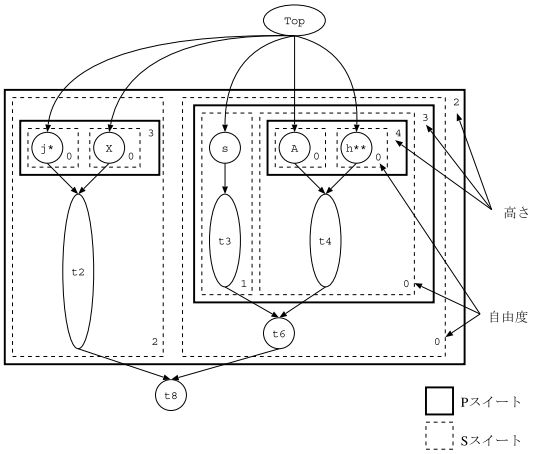

図 3 P スイートと S スイート
Fig. 3 Series suites and parallel suites.

P スイート（Parallel Suite） シリーズパラレル型レジスタ生存グラフにおいて，始点ノードと終点ノードが同じであるような，ノード，S スイートをひとまとめにしたもの．

高さ P スイートのネストの段数で定義される値で，ある P スイートの要素となっているノードや S スイートの高さとは，その P スイートのネストの段数のことである．

自由度 P スイートの各要素に対して定義される値で，次のように計算される．

(1) 対象とする P スイートに含まれるすべての要素について，それぞれ，最短生存時間を求める．ここで，ノードの最短生存時間とは，そのノードを実行するのに必要な時間の最小値のことである．また，S スイートの最短生存時間とは，その S スイートに含まれる要素それぞれの最短生存時間の和のことである．さらに，P スイートの最短生存時間とは，その P スイートに含まれる要素それぞれの最短生存時間の最大値のことである．

(2) 求まった最短生存時間の値の最大値となった要素の自由度を 0 とする．

(3) 他の要素の自由度はその要素の最短生存時間と，自由度 0 の要素の最短生存時間との差とする．

4. 本手法の詳細

本手法は，プログラム中の最内ループに適用され，その部分の命令レベル並列プロセッサ向けに最適化されたコード生成を行う．また，入力される中間コードは，SSA 変換を施された GPDG の形式になっているものとする．つまり，コード中の逆依存は取り除かれ，また，制御依存はガードの値の生産と使用のデータ依存に変換されているものとする．

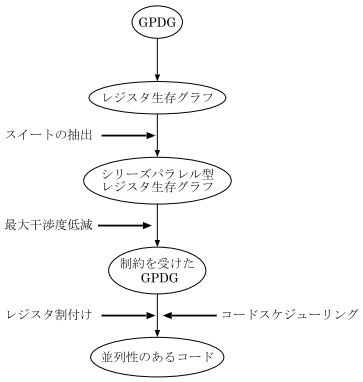

図 4 本手法におけるデータフロー
Fig. 4 Dataflow of our method.

本手法のデータフローを**図 4** に示す．

まず，入力された GPDG からシリーズパラレル型レジスタ生存グラフを生成する．次に，そのシリーズパラレル型レジスタ生存グラフにおいて，S スイート，P スイートを抽出する．その次に，抽出した S スイート，P スイートを用いて，生成されたシリーズパラレル型レジスタ生存グラフの最大干渉度低減操作を行う．最後に，最大干渉度を実レジスタ以下に低減させたシリーズパラレル型レジスタ生存グラフ上におけるレジスタの使用の時間順序を守るための依存を元の GPDG に加え，その GPDG に対してコードスケジューリングおよびレジスタ割付けを行う．

4.1 シリーズパラレル型レジスタ生存グラフの生成

最初に，GPDG からシリーズパラレル型レジスタ生存グラフを生成する手法について述べる．

まず，GPDG からレジスタ生存グラフを生成する．レジスタ生存グラフは，GPDG 中で使用されているすべての変数，つまりシンボリックレジスタをノードにし，そのシンボリックレジスタの値の使用に関するエッジを張ることで生成できる．これらのシンボリックレジスタを表すノードに加え，最内ループの入口となる TOP ノード，出口となる BOTTOM ノードを付加する．また，最内ループの入口ですでに生存しているシンボリックレジスタには，TOP ノードからエッジを張る．逆に最内ループの出口でも生存しているシンボリックレジスタからは，BOTTOM ノードに向けてエッジを張る．

図 1 の GPDG に対応するレジスタ生存グラフを**図 5** に示す．

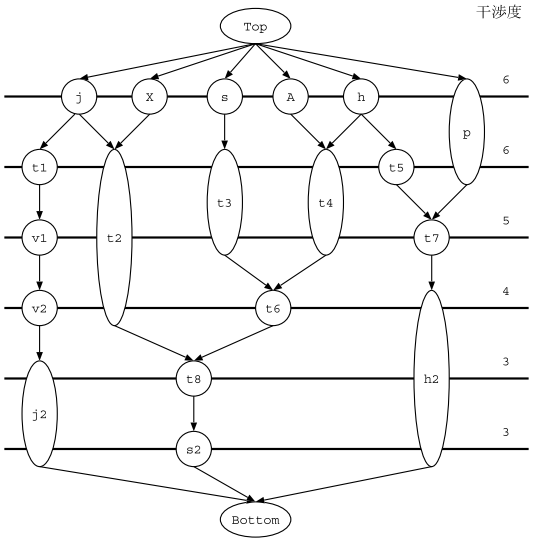

図 5 レジスタ生存グラフ
Fig. 5 An example of register existence graph.

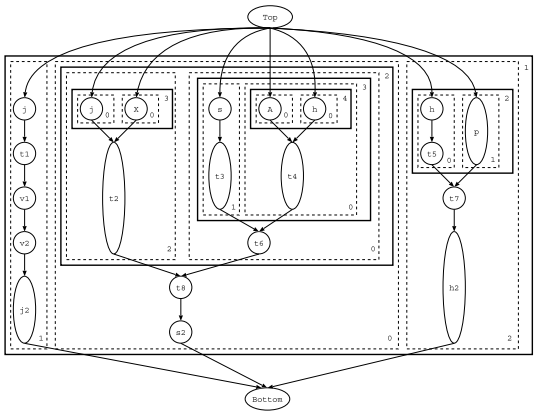

図 6 図 5 に対応するシリーズパラレル型レジスタ生存グラフ
Fig. 6 A series-parallelized register existence graph derived from Fig. 5.

次に，レジスタ生存グラフを S スイート，P スイートを抽出しながら，シリーズパラレル型のグラフに変形する．**図 6** に，図 5 のレジスタ生存グラフをシリーズパラレル型のグラフに変形し，P スイート，S スイートを抽出したものを示す．

4.2 最大干渉度低減操作のアルゴリズム

次に，シリーズパラレル型レジスタ生存グラフ上での，最大干渉度低減操作のアルゴリズムについて述べる．

まず，S スイート，P スイート，高さ，自由度の性質について述べる．

S スイートの性質 あるS スイートに注目したときに，その中で使用されている変数のうち，S スイートの入口と出口にあたる部分の変数以外は，その S スイート内のみでしか使われない．このことから，S スイートとはある一連のタスクで使用する変数の集まりであるといえる．したがって，同じ S スイート内の変数を連続的にレジスタに割り付けると，そのタスクは最大限円滑に実行されるようになると考えられる．

P スイートの性質 あるP スイートに注目すると，その中のすべてのスイートの出口で生成される変数を用いて，そのP スイートの次の変数値が生成されていることが分かる．このことから，P スイートは，ある時間に同時に計算がなされるべきスイートの組であるといえる．逆に考えると，P スイート内のスイートのうち 1 つでも実行が終わっていなければ，そのP スイートの実行を終えることはできないことになる．つまり，P スイート内のスイートは並列実行して，すべてのスイートの終了時間を早めるようにすることが重要となると考えられる．

高さの性質 高さとは，P スイートのネストの数であった．P スイート内のスイートを並列実行することを考えると，高さの高い部分ほど並列度が上がることになり，必要となるレジスタ数も増えると考えられる．また，あるP スイートの出口では，そのP スイートの最後で生成された変数値を利用して演算が行われている．つまり，あるP スイートの実行終了が遅れると，そのP スイートの最後で生成された変数の生存区間が伸びることにつながる．これら 2 つのことから，高さの高い部分には優先してレジスタの使用権を与え，コードスケジューリング時に実行の遅延が起きないようにすることが重要になると考えられる．

自由度の性質 自由度とは，あるP スイートの中で最も実行に時間がかかるスイートと，そのスイートの実行時間との差であった．あるP スイートに注目すると，そのP スイート内のすべてのスイートの実行が終了しないと次の演算が行えない．そのため，自由度の分だけ実行の遅延が起きてもそのP スイートよりも後ろの部分の実行開始時刻に対する影響はない．このことから，自由度は，あるP スイートの中でそのスイートの実行開始をどの程度遅延できるのかという指標になると考えられる．したがって，自由度の低いスイートほど優先して，レジスタの使用権を与えた方がよいと考えられる．

また，図 4 のデータフローから分かるように，本手

法では，最大干渉度低減操作の後にコードスケジューリングを行う．そのため，最大干渉度低減操作ではなるべく並列性を落とさないように考慮してレジスタの使用権を与えるようにしたい．そのため，本アルゴリズムでは，高さのほうが自由度よりも重要であると考える．

以上の性質を考慮した，最大干渉度低減操作のアルゴリズムは次のようになる．このアルゴリズムは，入力となるシリーズパラレル型レジスタ生存グラフから，最大干渉度を低減させたシリーズパラレル型レジスタ生存グラフを再構築する．

(1) 時刻の基準の決定
- (a) シリーズパラレル型レジスタ生存グラフにおいて，S スイート，P スイートの抽出を行う．
- (b) すべてのスイートのリストを作り，高さが高い順にソートする．
- (c) 同じ高さのスイートの自由度を求め，その自由度が低い順にソートする．
- (d) 同じ高さ，同じ自由度の場合，同じ P スイートに属する他のスイートのうち，以降の操作が完了していて，ノードの生存時刻が決定しているスイートの数の多い順にソートする．
- (e) ソートされたスイートのリストの先頭要素のノードの位置を，新たに作られる最大干渉度を低減したシリーズパラレル型レジスタ生存グラフ上に固定する．以降，この位置を基準にした相対的な時刻を使用する．
- (f) スイートのリストの先頭要素をリストから外す．

(2) 残りのノードの位置決定
- (a) スイートのリストの先頭要素を取り出す．これをカレントノードと呼ぶ．
- (b) 元のシリーズパラレル型レジスタ生存グラフにおける (1)-(e) で固定されたノードとカレントノードの相対的な時刻のずれを求める．
- (c) (2)-(b) で求まったずれと (2)-(e) で決まった新しいグラフ上での基準から求まる位置を A とする．また，すでに位置が定まっているノードとの依存関係から定まる位置を B とする．カレントノードの位置を A，B のうちより遅い方にする．このとき，その時刻の干渉度が実レジスタを超えていなければ，カレントノードをその位置に固定する．
- (d) もし，その時刻の干渉度が実レジスタを超えたら，スピルコードを挿入する．スピルインノードの位置は，同じ P スイート内の他のスイートの位置決めが終わっていない場合，可能な限り早い位置に挿入する．しかし，同じ P スイート内の他のスイートの位置決めが終わっている場合，それらのスイートの実行の終了位置に合わせ，可能な限り遅い位置に挿入する．
- (e) そのノードの位置決めが完了したことにより，ある P スイート内のすべてのスイートの位置決めが完了した場合，その P スイートの終点である接続点の位置を決める．
- (f) (2)-(a)〜(2)-(e) をリストから要素がなくなるまで繰り返す．カレントノードの位置決定の際，元々同じレジスタであったがシリーズパラレル型のグラフに変形する際にノードのコピーが起きたために作られた別ノードが同じ時刻に生存している場合には，そのノードとレジスタを

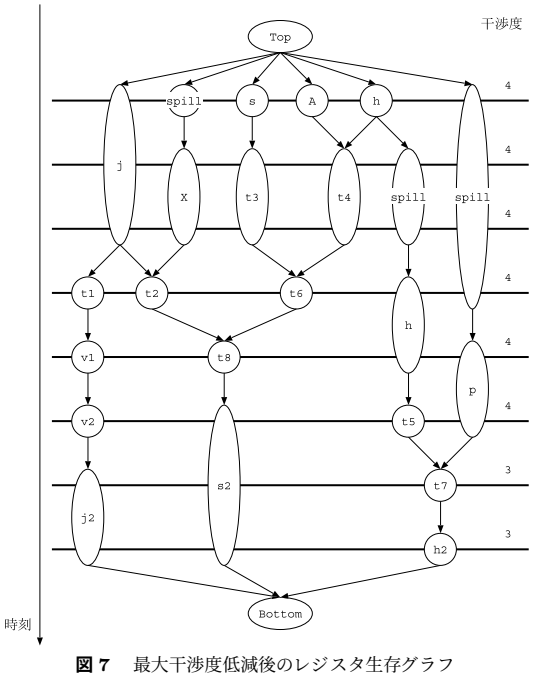

図 7 最大干渉度低減後のレジスタ生存グラフ
Fig. 7 A register existence graph after reduction of interference degree.

cycle	ALU1	ALU2	ALU3	ALU4
1	Load X	t4:=Load * (A+h)	Store h	
2	t3:=s≫1			
3	t1:=j+8	t2:= X+j	t6:=t3+t4	Load h
4	v1:=t1|1	t8:=t2+t6	Load p	
5	v2:=v1≪ 1	s2:=t8+2	t5:=h≫2	
6	j2:=v2+1	t7:=t5-p		
7	h2:=t7≪2			

cycle	REG1	REG2	REG3	REG4
–1	j	s	A	h
1–2	j	s	X	t4
2–3	j	t3	X	t4
3–4	t1	t2	t6	h
4–5	v1	t8	p	h
5–6	v2	s2	p	t5
6–7	j2	s2	t7	
7–8	j2	s2	h2	

図 8 生成されたコードとレジスタ割付け
Fig. 8 Generated code and its register allocation.

共有するようにする．

最大干渉度低減操作後のレジスタ生存グラフを，**図 7** に示す．なお，実レジスタ数を 4 として操作を行った．ノード j，h はシリーズパラレル型のグラフへの変形の際にそれぞれ 2 つのノードになって存在していたが，生存区間の重なりにより 1 つのノードにまとまっている．

4.3 コードスケジューリングおよびレジスタ割付け

最大干渉度低減の操作を行った後の，レジスタ生存グラフにおける，時刻の関係を保つために，元の GPDG に仮想依存エッジを張る．これにより，コードスケジューリングする際にレジスタが足りなくなることを避けることができる．

したがって，依存関係を保ちつつ，最大限 ALU 資源を生かしたコードスケジューリングを行えばよい．また，レジスタ制約を満たしているのでレジスタ割付けでは，適当に空いているレジスタを割り当ててやれば完了する．

図 7 の状態を反映させてコードスケジューリングおよびレジスタ割付けした結果を**図 8** に示す．

5. 評　　価

本手法と従来手法の比較のために，スタンフォードベンチマーク☆中で用いられているいくつかの関数の最内ループに，本手法を机上シミュレーションして評価を行った．なお，評価対象としたプログラムの中には関数呼び出しは含まれていなかった．

対象とするプロセッサは，整数演算，ストア命令は 1 サイクル，ロード命令は 2 サイクルで実行できる VLIW アーキテクチャとし，レジスタ数が 8，32，プロセッサの並列度が 2，4，8，$+\infty$ の場合について評価をとった．なお，このレジスタ数に，アドレスレジスタは含まない．

用いた手法は，レジスタ干渉グラフとグラフカラーリングを用いた手法（手法 1），並列化レジスタ干渉グラフとグラフカラーリングを用いた手法（手法 2）と本手法である．評価は，生成されたコードのクリティカルパス長によって行った．クリティカルパスの長さはプログラムの実行時間に相当するため，この数値の比較によりプログラム速度の比較を行うことができる．**表 1**，**表 2** に結果を示す．

手法 1 であるが，これは並列性を考慮していないため，プロセッサの並列度が上がっても，顕著な性能の向上が見られていない．

手法 2 では，並列性を考慮したレジスタ干渉グラフを用いることにより並列性の抽出を行っているが，コードの並列度に対するレジスタ数が少ないときには，レジスタプレッシャーが上がる結果となり，余分なスピルコードが出てしまうことがある．**図 9** と**図 10** に，関数 Bubble 中の最内ループの 1 つの基本ブロックに対する，本手法と手法 2 によるコードスケジューリングおよびレジスタ割付けの結果を載せる．両結果とも並列性の抽出には成功しているが，手法 2 では余分なスピルコードによりクリティカルパスが伸びてしまい，性能の向上の妨げとなっている．手法 2 では，並列性の抽出をコード全体に対して見いだすようにしているため，レジスタの生存時間を長く見積もる結果となることが多く，このような余分なスピルコードの増加が避けられない．

これに対し，本手法では並列性を見いだす部分を，パラレルスイートによりまとめているため，その部分に対する並列度の抽出の妨げになるような割付けは行われない．また，シリーズスイートにより，レジスタを使い回すことによって妨げられないタスクの列を見いだすことにより，余分なスピルコードの挿入を避けることに成功している．

全体をみても，本手法により生成されたコードの方が速度が向上している．Fit のように，コード中の並列性が低いものや，Qsort の並列度が 2 の場合のようにプロセッサの持つ並列性が低い場合，レジスタ数が

☆ スタンフォードベンチマークは，1988 年にスタンフォード大学の J. Hennessy と P. Nye によって，RISC や CISC マシンのパフォーマンスを評価するために開発された，行列演算やソートなどの比較的小さなプログラムからなるベンチマークプログラムである．

表 1 評価（レジスタ数 8）
Table 1 Critical path length (with 8 registers).

手法	手法 1				手法 2				本手法			
並列度	2	4	8	∞	2	4	8	∞	2	4	8	∞
Bubble	37	37	37	37	31	18	17	17	23	15	13	13
Exptab	91	91	91	91	48	42	23	23	44	24	17	17
Fit	77	75	75	75	149	147	147	147	77	75	75	75
Initarr	96	96	96	96	83	55	55	55	49	33	33	33
Initmat	110	110	110	110	76	73	73	73	57	34	30	30
Permute	25	24	24	24	39	28	28	28	20	17	17	17
Qsort	34	34	34	34	44	30	27	27	34	24	21	21

表 2 評価（レジスタ数 32）
Table 2 Critical path length (with 32 registers).

手法	手法 1				手法 2				本手法			
並列度	2	4	8	∞	2	4	8	∞	2	4	8	∞
Bubble	26	24	23	23	23	14	13	13	23	14	13	13
Exptab	38	33	33	33	35	19	11	10	35	19	11	10
Fit	73	72	72	72	73	72	72	72	73	72	72	72
Initarr	51	49	49	49	39	33	33	33	39	33	33	33
Initmat	62	58	58	58	51	30	28	28	49	30	28	28
Permute	25	23	23	23	20	17	17	17	20	17	17	17
Qsort	28	23	23	23	28	22	20	20	28	22	20	20

cycle	ALU1	ALU2	ALU3	ALU4
1	mem1:=Store(r1)	t20:=r0+32767	t40:=r1|0	t42:=r0+4
2	t18:=r1|0	t21:=t20&32767	t41:=t40>>2	t43:=t42&4
3	t19:=t18>>2	t44:=t41+t43	t30:=r1|0	t32:=r0+4
4	t22:=t19+t21	t31:=t30>>2	t33:=t32&4	t28:=r0+32767
5	REG8:=Load(mem1)	t34:=t31+t33	t29:=t28&32767	t38:=r0+32767
6	t26:=r0+32767	t35:=t34+t29	t39:=t38&32767	t23:=Load(t22)
7	t24:=r1|0	t27:=t26&32767	t36:=Load(t35)	t45:=t44+t39
8	t25:=t24>>2	*(t45):=Store(t23)	REG1:=Load(mem1)	
9	t37:=t25+t27			
10	*(t37):=Store(t36)	t46:=r1+1		

cycle	REG1	REG2	REG3	REG4	REG5	REG6	REG7	REG8
–1	r1							
1–2	r1	t20	t40	t42				
2–3	t18	t21	t41	t43				
3–4	t19	t21	t30	t32	t44			
4–5	t22	t21	t31	t33	t44	t28	t38	
5–6	t22	t21	t34	t29	t44	t28	t38	r1
6–7	t22	t21	t35	t39	t44	t23	t26	r1
7–8	t22	t24	t36	t45	t44	t23	t27	r1
8–9	r1	t24	t36	t45	t44	t23	t27	t25
9–10	r1	t37	t36	t45	t44	t23	t27	t25
10–	t46	t37	t36	t45	t44	t23	t27	t25

図 9 本手法により生成されたコード
Fig. 9 Generated code for a basic block of Bubble via our method.

潤沢にある場合には，従来手法と同程度の速度になっている．このことから，並列性の抽出が難しい場合や，従来手法の性能が生かされるような状況でも従来手法と同等の性能を発揮できることが分かる．また，レジスタ数 8 のときと 32 のときの結果の差違が他の手法に比べて小さい．このことは，本手法がレジスタプレッシャーが高い場合でも性能を維持できることを示している．

cycle	ALU1	ALU2	ALU3	ALU4
1	t30:=r1|0	t32:=r0+4	t26:=r0+32767	t42:=r0+4
2	t31:=t30>>2	t33:=t32&4	t27:=t26&32767	t43:=t42&4
3	t34:=t31+t33	t28:=r0+32767	mem1=Store(t27)	t38:=r0+32767
4	t29:=t28&32767	t40:=r1|0	t39:=t38&32767	
5	t35:=t34+t29	t24:=r1|0	t41:=t40>>2	REG6:=Load(mem1)
6	t36:=Load(t35)	t25:=t24>>2	mem2:=Store(t41)	t18:=r1|0
7	t37:=t25+t27	t20:=r0+32767	REG3:=Load(mem2)	t19:=t18>>2
8	*(t37):=Store(t36)	t21:=t20&32767	t46:=r1+1	
9	t22:=t19+t21	t44:=t41+t43		
10	t45:=t44+t39	t23:=Load(t22)		
11	nop			
12	*(t45):=Store(t23)			

cycle	REG1	REG2	REG3	REG4	REG5	REG6	REG7	REG8
–1	r1							
1–2	r1	t32	t30	t42	t26			
2–3	r1	t33	t31	t43	t27			
3–4	r1	t34	t28	t43	t27		t38	
4–5	r1	t34	t29	t43	t40		t39	
5–6	r1	t35	t24	t43	t41	t27	t39	t18
6–7	r1	t36	t25	t43	t41	t27	t39	t19
7–8	r1	t36	t41	t43	t37	t20	t39	t19
8–9	t46	t36	t41	t43	t37	t21	t39	t19
9–10	t46	t36	t41	t22	t44	t21	t39	t19
10–11	t46	t36	t41	t23	t44	t21	t45	t19
11–12	t46	t36	t41	t23	t44	t21	t45	t19
12–	t46	t36	t41	t23	t44	t21	t45	t19

図 10 手法 2 により生成されたコード
Fig. 10 Generated code for a basic block of Bubble via method2.

6. おわりに

本稿では，シリーズパラレル型レジスタ生存グラフを用いたレジスタ割付け手法の改善について述べた．本手法では，まずシリーズパラレル型レジスタ生存グラフを作り，スイートを抽出する．そして，そのスイートの高さによって，並列性のある場所を見つけ，優先的にレジスタへの生存権を与えている．その結果をコードスケジューラに引き継ぐことによって，並列性を損なわないようなコードスケジューリングが可能になった．

今後は，上で述べたような，最大干渉度低減アルゴリズムの改善と，シリーズパラレル型のレジスタ生存グラフの効率的な作成法について研究を進めていきたいと考えている．

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) Norris, C. and Pollock, L.L.: A Scheduler-Sensitive Global Register Allocator, *Proc. ACM SIGPLAN '93 Conference on Supercomputing*, pp.804–813 (1993).
3) Pinter, S.S.: Register Allocation with Instruction Scheduling: A New Approach, *Proc. ACM SIGPLAN '93 Conference on Programming Languages Design and Implementation*, pp.248–257 (1993).
4) 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及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).
5) Cyton, R., Ferrante, J., Rosen, B., Wegman, M. and Zadeck, K.: An Efficient Method of Computing Static Single Assignment Form, *Conference Record of the 16th ACM Symposium on the Principles of Programming Languages*, pp.25–35 (1989).
6) 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).
7) 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).
8) Duffin, R.J.: Topology of Series-Parallel Networks, *J. Math. Applic.*, 10, pp.303–318 (1965).
9) Zhou, X., Suzuki, H. and Nishizeki, T.: A Linear Algorithm for Edge-Coloring Series-Parallel Multigraphs, *J. Algorithms*, No.20, pp.174–201 (1996).

(平成 12 年 1 月 13 日受付)
(平成 12 年 9 月 7 日採録)

近藤　伸宏

1974 年生．1997 年早稲田大学理工学部情報学科卒業．1999 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．同社研究開発センターにおいて，コンパイラの研究に従事．

古関　　聰（正会員）

1969 年生．1994 年早稲田大学大学院理工学研究科電気工学専攻修士課程修了．1998 年同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．

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（情報科学）．

深澤　良彰（正会員）

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中退．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．ソフトウェア科学会，IEEE，ACM 各会員．