条件分岐を考慮した
ソフトウェア・パイプラインにおける
レジスタ割付の研究

工学研究科
筑波大学

2002年3月

糸賀 裕弥

条件分岐を考慮した
ソフトウェア・パイプラインにおける
レジスタ割付の研究

Register Allocation Methods for Software Pipelined Loops with Conditional Branches

糸賀 裕弥

2002年3月

概 要

本論文では，条件分岐を含むループにおけるソフトウェア・パイプライン処理に着目し，これに対するレジスタ割り付け方法について検討を行なった．

述語付き命令実行機構によるハードウェア補助が無いプロセッサにおいて，条件分岐を含むループのソフトウェア・パイプラインを行なう効果的な方法としてEnhanced Modulo Scheduling（EMS）及び改良EMSがある．まず，EMS及び改良EMSに対する干渉グラフを用いたレジスタ割り付け方法として，状態別干渉グラフと状態合併干渉グラフを用いた方法を提案する．レジスタ割り付け実験において，正確であるがグラフが巨大となる状態別干渉グラフに対して，グラフを縮退させた状態合併干渉グラフは，まれに不利益が生じるにもかかわらず彩色数に遜色はなく，高速にレジスタ割り付けが行なえることが判明した．

次に，レジスタ改名機構向けに提案されたSpiral Graphを，述語付き命令実行機構を合わせもつIA-64のようなアーキテクチャに対して拡張した述語付きSpiral Graphと述語付きShort Bridge Algorithmを提案する．同時に存在する変数は，通常別のレジスタに割り付けられるが，述語が異なる命令で使われる変数は同一のレジスタに割り付けることができる．同一のレジスタへの割り付け方針をレジスタ割り付け実験において比較したところ，変数の隙間を小さくするShort Bridge Algorithmの原理が有効であることが判明した．

もくじ

第1章 序論 **1**
1.1 研究の背景 1
1.2 研究の目的 4
1.3 本論文の構成 5

第2章 基礎知識 **8**
2.1 ソフトウェア・パイプライン 8
2.2 モジュロ・スケジューリング 8
2.3 レジスタ改名機構 9
2.4 干渉グラフと彩色アルゴリズム 10
2.4.1 レジスタ干渉グラフ 10
2.4.2 スライドレジスタ干渉グラフ 11
2.4.3 簡単な彩色近似アルゴリズム 12
2.4.4 k-彩色近似アルゴリズム 12
2.5 Spiral Graph と Short Bridge Algorithm 12
2.5.1 Spiral Graph 12
2.5.2 Short Bridge Algorithm 13
2.6 述語付き命令実行機構 14

第3章 述語付き命令実行機構を持たないアーキテクチャにおけるレジスタ割付法 **15**
3.1 EMS とその改良法 15
3.1.1 条件分岐を含むコードに対するソフトウェア・パイプライン法 15
3.1.2 EMS によるスケジュール 18
3.1.3 改良 EMS によるアプローチ 19
3.1.4 改良 EMS による命令スケジュール 20
3.2 EMS の実行モデルの解析 21
3.3 干渉グラフを用いたレジスタ割り付け方法 23
3.3.1 状態別干渉グラフによるレジスタ割り付け 24
3.3.2 状態合併干渉グラフによるレジスタ割り付け 26
3.3.3 グラフの重ね合わせによる問題点 28

3.4 Spiral Graph を用いたレジスタ割り付け方法 30
3.5 レジスタ割り付け実験 31
3.5.1 実験の目的 31
3.5.2 実験方法 31
3.5.3 ベンチマークによる評価結果 32
3.5.4 自動生成プログラムによる評価結果 33
3.6 レジスタ割り付け実験の考察 37
3.7 関連研究 37

第4章 述語付き命令実行機構を持つアーキテクチャにおけるレジスタ割り付け法 40
4.1 干渉グラフを用いたレジスタ割り付け方法 40
4.2 Spiral Graph を用いた方法 41
4.2.1 レジスタ改名機構と述語付き命令実行機構をあわせもつアーキテクチャ 41
4.2.2 Spiral Graph と述語付き命令実行機構 42
4.2.3 述語付き Spiral Graph 43
4.2.4 述語付き Short Bridge Algorithm 44
4.2.5 レジスタ共有方針 45
4.3 レジスタ割り付け実験 47
4.3.1 実験の目的 47
4.3.2 実験方法 47
4.3.3 レジスタ数の下界との比較 48
4.3.4 実験結果 49
4.4 レジスタ割り付け実験の考察 51
4.5 関連研究 52

第5章 結論 54
5.1 述語付き命令実行機構を持たないアーキテクチャにおけるレジスタ割付法 54
5.2 述語付き命令実行機構を持つアーキテクチャにおけるレジスタ割付法 55
5.3 今後の展開 56

参考文献 57

謝辞 66

表 目 次

3.1 LFK #16に対する彩色結果 33
3.2 彩色アルゴリズム1（coloring）による彩色数の差 34
3.3 彩色アルゴリズム1（coloring）で彩色に要したCPU時間の平均 . . 34
3.4 グラフの頂点数の比較 35
3.5 彩色アルゴリズム2（k-coloring）による彩色数の差 35
3.6 彩色アルゴリズム2（k-coloring）で彩色に要したCPU時間の平均 . 36
3.7 彩色アルゴリズムどうしの比較 36

図 目 次

2.1 ローテイティング・レジスタ 9
2.2 レジスタ干渉グラフ 10
2.3 スライドレジスタ干渉グラフ 11
2.4 Spiral Graph 13
2.5 Short Bridge Algorithm 13

3.1 一般化された条件分岐構造を含む多重ループ 16
3.2 サンプルプログラム 17
3.3 改良 EMS におけるモジュロ・スケジュール結果 20
3.4 改良 EMS における逆 IF 変換結果 22
3.5 状態遷移の様子 23
3.6 状態別干渉グラフ 25
3.7 状態合併干渉グラフ 27
3.8 重ね合わせによって色数が増えてしまう場合 29
3.9 改良 EMS のために拡張された Spiral Graph 30
3.10 Livermore Fortran Kernel #16. 32

4.1 述語付き Spiral Graph 44
4.2 chains/case sensitive lower bound. 49
4.3 chains/yamashita's lower bound. 49
4.4 trees/case sensitive lower bound. 50
4.5 trees/yamashita's lower bound. 50
4.6 random/case sensitive lower bound. 51
4.7 random/yamashita's lower bound. 51

第1章　序論

1.1　研究の背景

近年，計算機の処理速度に対する要求がさらに高まっている．

計算物理学（Computational Physics）と呼ばれる分野においては，従来からの実験物理学では実験や再現が困難である事象を，計算機上で仮想的にシミュレーションすることで，実際とほぼ同等の結果を得ることができるようになっている．計算物理学の手法は，素粒子物理学，物性物理学，宇宙物理学など物理学の幅広い分野にわたっている．計算物理学において，実際の現象をさらに精緻に模倣していくためには，より高い計算処理能力が求められる．

計算物理学などの高性能科学技術計算のために，筑波大学と日立製作所が共同で開発した分散メモリ型超並列計算機CP-PACS（SR-2201）[NNB96, NYO96]は，単体でのピーク性能が300MFLOPS（Mega FLoating-point Operations Per Second）の単体プロセッサ2,048台を有し，超並列計算機全体として600GFLOPSのピーク性能を持っている．CP-PACS開発当時の1996年には，スーパーコンピュータの実効速度ランキングであるTOP500 List[Top]において，368.2GFLOPSの世界最速性能を持つと認定された．

CP-PACSが対象としている科学技術計算には，数週間から数ヶ月にわたって連続実行されるループプログラムも存在する[NNB96]．このような計算においては，コンパイラによるコードの最適化が計算時間に及ぼす影響が大きい．

計算機の処理速度への期待が高まっているのは，スーパーコンピュータによる科学技術計算の分野だけではない．一般に普及しているパーソナルコンピュータにおいても，より高い計算処理能力への要求が増大し続けている．このため，従来はスーパーコンピュータのアーキテクチャとして提案および実装されていた機構，例えばスーパースカラなどの命令の同時実行機能などがパーソナルコンピュータ向けのプロセッサにおいても実装され利用されるようになってきている．

CP-PACSの単体プロセッサだけでなく，現在一般のPCに普及しているプロセッサにおいても，単体プロセッサの演算性能を高めるために，**命令レベル並列性**を抽出できる**命令スケジューリング**手法や，レジスタ溢れを起こしにくい**レジスタ割り付け**手法が必要となっている．

ソフトウェア・パイプライン[AJLA95]は，命令レベル並列性[RF93]を利用したループプログラム最適化手法である．実行される命令をコンパイル時に適切に並

べ換え，命令と命令の間の待ち時間に，別の繰り返しの命令を実行することでパイプライン処理を行なう．命令の並べ換えを行なうことを命令スケジューリングと呼ぶ．パイプライン処理により，繰り返し１回ごとの開始間隔が短かくなるので，ループプログラムの総実行時間を短かくすることができる．

例えば複数の命令を同時に実行できるスーパースカラアーキテクチャでは，浮動小数点数演算など実行に時間がかかる命令の間に，別の繰り返しにおける主記憶からの値のロード命令を実行することによって，もとの浮動小数点数演算命令の待ち時間を有効に活用することができる．さらに，実際に使う繰り返しよりも１回前の繰り返しで値のロード命令を発行できることから，相対的に応答速度の遅い主記憶による遅延を隠蔽することにも利用できる．

基本的なループに対するソフトウェア・パイプラインを用いた最適化は，すでに一般的に普及した技術であるが，ソフトウェア・パイプラインは原理的に２つの問題をかかえており，これらの場合における適用方法については現在も研究が進められている [WLmWH93]．

一つは，命令の並べ換えによってソフトウェア・パイプライン処理を行なおうとしたときに，パイプライン処理のために重ね合わせられた複数の繰り返しが，同じレジスタを使って異なる繰り返しで使われる値を保持できないことである．すなわち i 回目の繰り返しにおいて使われる変数と，$i+1$ 回目の繰り返しの変数は，通常のプログラムでは同じ場所，同じレジスタを用いて値を保持することができるが，i 回目と $i+1$ 回目の繰り返しが部分的に重ね合わせられて実行されるソフトウェア・パイプラインでは，同時に異なる値を保持する必要があることから同じ場所，同じレジスタを用いることができない．

もう一つは，条件分岐を含むプログラムをソフトウェア・パイプライン処理によって高速化できるかである．i 回目の繰り返しと $i+1$ 回目の繰り返しを部分的に重ね合わせて実行するためには，i 回目，$i+1$ 回目の繰り返しにおける条件判定の結果によって，i 回目の繰り返しで then 節あるいは else 節の命令を，同様に $i+1$ 回目の繰り返しにおいても then 節あるいは else 節の命令のいずれかを組み合わせて実行する必要がある．これをコンパイル時にあらかじめ決定することはできない．

プログラム中の変数が生成されてから参照されなくなるまでの区間を変数の**生存区間（live range）**と呼ぶ．ソフトウェア・パイプラインにおける問題点の前者，すなわち i 回目の変数の生存区間が，$i+1$ 回目の同じ名前の変数の生存区間と重なりあってしまう問題に対する，ハードウェアを用いた解決方法が**レジスタ改名（register renaming）機構**である．

CP-PACS の単体プロセッサにおいては，**スライドウィンドウ・アーキテクチャ（Slide-Window Architecture）** [NNB96, INBN93] と名付けられたレジスタ改名機構がハードウェアとして用意された．従来ベクトルレジスタで行なっていた演算とほぼ同じ処理を，スライドウィンドウ・アーキテクチャによるソフトウェア・パイプライン処理で，次々とメモリからレジスタへのロードを行ないつつ演

算を行なう擬似ベクトル処理として実現した [Nak95, INBN93]．実行中にレジスタ番号を適切に変更することで，ある変数の生存区間が自身の生存区間と重ならないようにする，という命令スケジューリングにおける制約を緩和することができる．さらに，ループの数回・数十回前の繰り返しにおいてロード命令を発行できることや，スライドウィンドウの外側に存在する物理レジスタにあらかじめ値をロードしておく命令を利用することで，キャッシュが有効でない広範囲のデータを，相対的に応答速度の遅い主記憶から適切に得ることができる利点がある．

このため，CP-PACS の単体プロセッサ向け最適化においては，スライドウィンドウ・アーキテクチャへの適切なレジスタ割り付け方法の確立と，ソフトウェアによる条件分岐を含むループに対する最適化方法の確立に重点が置かれている [NYO96]．

基本的なループにおけるレジスタ割り付け方法としては，**スライドレジスタ干渉グラフ**および**Spiral Graph**による方法が提案されている [HSYN98, Har00] が，条件分岐を含む場合への適用方法が問題となる．

インテルが開発した **IA-64 アーキテクチャ**は，Itanium プロセッサとして実装され一般に普及し始めている．現在は一般のパーソナルコンピュータより処理速度が要求される小型サーバの分野での普及が期待されている．

IA-64 アーキテクチャにおいても，CP-PACS の単体プロセッサと同様に，変数の生存区間による命令スケジューリングにおける制約を緩和するためのレジスタ改名機構が採用されている．IA-64 アーキテクチャにおけるレジスタ改名機構は，スーパーミニコンピュータ Cydra5[BYA93] において提案され実装された**ローテイティング・レジスタ（Rotating Register）**の応用である．

さらに IA-64 アーキテクチャにおいては，ソフトウェア・パイプラインにおけるもう一つの問題である条件分岐構造の実行を補助するためのハードウェア機構として，**述語付き命令実行機構（Predicated Execution）**があわせて実装されている．条件分岐構造を述語付き命令に変換することにより，複数の実行パスを持つプログラムを単一の実行パスのプログラムに変換することができる．

IA-64 アーキテクチャにおいては，ローテイティング・レジスタおよび述語付き命令実行機構がソフトウェア・パイプラインによる実行を補助するものとして採用されている．コンパイラはこれらを適切に利用し最適化を行なう必要があるが，最適化のための分析や定式化が十分であるとはいえない．

スライドウィンドウ・アーキテクチャ向けに提案された Spiral Graph は，ローテイティング・レジスタにおいてもほぼそのまま利用可能である．従来のレジスタ割り付け法において使われるグラフよりもレジスタ改名機構に対する定式化が素直であることや，Short Bridge Algorithm による割り付け結果が良好なこと，さらに特定の条件においては多項式時間で最適な割り付け結果を得られるなどの利点があるが，そのままでは述語付き命令実行機構を表現することが困難であることが問題となる．

1.2 研究の目的

本研究では，条件分岐を含む場合のソフトウェア・パイプラインにおけるレジスタ割り付けを最適に行なうことを目的としている．

レジスタは，プロセッサ内に存在する，非常に高速であるが，数の限られた小容量のメモリである．ロード・ストアアーキテクチャを採用した一般のRISCプロセッサにおいては，すべての演算命令はレジスタに格納された値に対して行なわれる．それ以外のアーキテクチャにおいても，レジスタの値を利用した命令は実行が非常に高速であることが多い．プログラム中で用いられる変数や，演算の中間結果を保持するために使わる一時変数（以降これらをまとめて**変数**と呼ぶ）はこの小容量のレジスタに格納すると都合が良い．

数の限られたレジスタに対して，プログラム中の変数を対応付けることをレジスタ割り付けという．

的確なレジスタ割り付けによって，高速な命令を使えるようになるほか，相対的に応答速度の遅い主記憶への中間結果の退避・回復処理やレジスタどうしのデータの移し換えなどの処理を，除去あるいは減少させることができる．有限個のレジスタを効率良く使うことで，プログラムの性能を向上させることができる．

従来のレジスタ割り付けにおいては，対象とする目的コードに必要なレジスタ数が，アーキテクチャによって規定された利用可能なレジスタ数よりも少なくなるように割り付けることを目的としていた．割り付けの結果，もし利用可能なレジスタ数が，必要とするレジスタ数よりも少ない場合には中間結果の主記憶への退避・回復処理を行なう必要がある．この処理が少ないものをより最適であると考えると，レジスタ数が足りている場合にはレジスタ割り付け結果は常に最適解であると言える．

本研究におけるレジスタ割り付け結果の最適性は，これよりも厳しく，プログラムの実行に必要なレジスタ数を最適に見積もり，必要最小限の数での割り付けを行なうことを表わすことにする．レジスタ数の最小化は，よりパイプライン段数の多いソフトウェア・パイプラインの命令スケジューリングや，さらなるループ展開などコード最適化への可能性を広げる．つまり，合計32本のレジスタが利用できるとき，必要レジスタが17であると見積もられるならばアーキテクチャの規定するレジスタ数を下回っており，レジスタの退避・回復などによる性能低下を招くことなく実行が可能である．ところで，見積もりをより精緻に行ない必要レジスタが16で十分とわかったならば，それまでの2倍のループ展開を行なうことで，ループ制御に必要なループバックジャンプの回数を減少させることができる．すなわち必要レジスタ数の最小化によって，32本のレジスタすべてを使ってさらなる最適化が可能となる．

本研究で対象としている条件分岐を含むプログラムは，多重ループの最内ループに条件分岐を含むようなプログラムである．なお，条件分岐構造が多重化されていてもよい．ループを形成するためのループ制御変数の境界チェックなどは，ルー

プの外側にあるものとして，ここでは扱わない．

条件分岐を含むプログラムにおいては，条件が成立した場合と，条件が不成立だった場合に実行されるコードが異なる．プログラム中の変数の定義・使用は実行される命令によって規定されるから，条件判定の結果によって異なる実行パスごとにそこに存在する変数・使用の挙動も全く異なり，かつこれらの実行パスどうしは同時に実行されることが無い．

レジスタは計算途中のデータの保持に用いられるから，複数の実行パスに存在する変数を矛盾なくプロセッサ内のレジスタに割り付ける必要がある．さらに，条件が成立した場合にのみレジスタへの保持が必要となる変数は，条件が成立しなかった場合には保持が不要となる．もちろんその逆の場合もある．条件判定によって不要となる値を保持するためのレジスタは，別の目的に利用することができる．通常は別の実行パスにおいて必要な値を保持するために，別の変数に割り付けられる．実行パスによって異なる変数に１つの実レジスタが割り付けられ矛盾なく利用されることを本論文ではレジスタを共有すると呼ぶことにする．

本研究では，異なる実行パスにおいて共有可能なレジスタを適切に組み合わせ，レジスタ数を最小化することを行なっている．

なおコード生成には，命令スケジューリングとレジスタ割り付けのうち，命令スケジューリングを先に行ない，のちにレジスタ割付を行なうプリパス・スケジューリング（Prepass Scheduling）と，レジスタ割付を先に行ない，のちに命令スケジューリングを行なうポストパス・スケジューリング（Postpass Scheduling）がある [Lam88] が，本研究ではプリパス・スケジューリングを採用している．これは，一般的に命令スケジュールを行なったほうが命令の並列度を上げることができ実行性能の面で有利だからである．

1.3 本論文の構成

本論文の構成は以下の通りである．

第２章では，まず，基礎知識としてループの最適化法であるソフトウェア・パイプライン法について，一般的に使われる命令スケジュール方法であるモジュロ・スケジュールについて説明する．さらに，ソフトウェア・パイプライン実行を支援するために導入されたハードウェア機構である，レジスタ改名機構について述べる．

これに対する，レジスタ割り付け手法として

1. 干渉グラフと彩色アルゴリズムを用いたレジスタ割り付け方法 [CAC+81, Cha82]
2. Cyclic Interval Graph と Fat Cover Algorithm を用いたレジスタ割り付け方法 [HGAM92a, HGAM92b]

と，これらの２つの方法に対するレジスタ改名機構向けの拡張法である

1. スライド干渉グラフとスライド彩色アルゴリズムを用いたレジスタ割り付け方法 [HSYN98]
2. Spiral Graph と Short Bridge Algorithm を用いたレジスタ割り付け方法 [HSYN98, Har00]

について用語を確認しながら詳しく述べる.

最後に，条件分岐構造を含むプログラムの実行を支援するために導入されたハードウェア機構である，述語付き命令実行機構について述べる.

第3章では，現在普及している一般的なプロセッサに対する条件分岐を含むループのソフトウェア・パイプライン法として EMS（Enhanced Modulo Scheduling）及び改良 EMS に注目する.

EMS 及び改良 EMS における複数のパイプライン・カーネル間の状態遷移モデルの解析を行ない，これらのカーネルに干渉グラフを用いたレジスタ割り付け方法を適用する方法について，状態別干渉グラフと状態合併干渉グラフの 2 つの手法を提案する.

状態別干渉グラフを用いた方法では，すべてのパイプライン・カーネルの干渉グラフを生成し，その間のノードの同一性を制約として用いる.

状態合併干渉グラフを用いた方法は，状態別干渉グラフにおいて問題となるグラフの巨大化の問題を解決するために，カーネルごとのグラフを重ね合わせて縮退させる方法である.

こららのグラフのそれぞれの利点，問題点について述べ，性能評価実験の結果について述べる.

第4章では，レジスタ改名機構と述語付き命令実行機構をあわせ持つアーキテクチャにおける Spiral Graph を用いたレジスタ割り付け法について述べる．レジスタ改名機構と述語付き命令実行機構は，Intel IA-64 アーキテクチャ[Int99, Ike00, Jin99]において採用され，今後同アーキテクチャの利用が拡大すると考えられる.

Spiral Graph 及び Short Bridge Algorithm はレジスタ改名機構を持つアーキテクチャに対して非常に有効であるが，条件分岐を含む場合への適用方法について詳しく解析されていなかった．そこで述語付き命令実行機構を持つ場合について，これらを拡張した述語付き Spiral Graph と述語付き Short Bridge Algorithm を提案する.

条件分岐構造の実行に述語付き命令を用いると，述語の値によっては，同時に存在する複数の変数が単一の実レジスタを共有することが可能となる．述語付き Spiral Graph では，実レジスタを表現するトラックを複数の副トラックに拡張した.

単一の実レジスタを共有可能な変数の生存区間は複数存在するから，実レジスタの共有戦略，すなわち副トラックへの割り付け方針について

1. exclude allocation
2. joint allocation
3. include allocation

4. look-ahead allocation
5. slide-cover allocation

の5つの方法を用意した.

これらの戦略のそれぞれの利点，問題点について述べレジスタ数の下限との比較評価実験の結果について述べる.

最後に，第5章において，本研究のまとめと研究の今後の展開について述べる.

第2章　基礎知識

2.1　ソフトウェア・パイプライン

ソフトウェア・パイプライン [AJLA95, Lam88] は，命令レベル並列性を利用したループ最適化法である．

プログラムのコンパイル時に，ループの繰り返しを互いに一定の間隔で重ね合わせたコードを生成することで，実行の高速化を行なう．例えば，i 番目の繰り返しに $i+1$ 番目の繰り返しの一部を，さらに $i+2$ 番目の繰り返しの一部をおり混ぜたコードを生成しておく．これらの繰り返しの一部を i 番目の繰り返しと区別するために，もとのループから何回離れた繰り返しの命令であるかを相対的な**パイプライン・ステージ番号**として表現する．実行時には，ループの繰り返しをそのまま実行するよりも短かい間隔で繰り返しが実行されていくため，ループの総実行時間は短かくなる．

ソフトウェア・パイプラインのために命令スケジュールされたコードは大きく3つの部分に分けられる．ループが定常状態になりすべてのステージに命令が充填されている場合のコードを，ソフトウェア・パイプラインのカーネルまたは単に**カーネル**と呼ぶ．ループ処理が始まり定常状態に以降するまでをプロローグ，逆に定常状態からループを完全に脱出するまでのコードをエピローグと呼ぶ．

2.2　モジュロ・スケジューリング

モジュロ・スケジューリング [Nak99] は次の手順でスケジュールを行なう．

1. ソフトウェア・パイプライン処理の性能を決定するループ立ち上げ間隔（II：Initiation Interval）の最小値を決定する．
2. 最小の II から決まる資源予約表（Resource Reservation Table）に対して，命令間のレイテンシあるいは依存関係を満たしつつ命令を配置する．
3. もし命令間のタイミングやレジスタ割付不能となってしまった場合には，II を増やして再度スケジュールを行なう．

与えられたプログラムの構造とプロセッサに存在する演算リソースから求められる最小の II から命令スケジュールを試していく．スケジュール方法は単純であるが，ソフトウェア・パイプライン処理の性能を決める要因である II を小さく抑

えた命令スケジュールを得やすいという特徴を持ち，一般的によく使われている手法である．

2.3 レジスタ改名機構

レジスタ改名機構はCydra5[BYA93]にローテイト・レジスタとして提案及び実装されたソフトウェア・パイプライン向けのハードウェアで，添え字であるレジスタ番号をレジスタの内容を保持したまま一斉に変更することができる．これによりデータのWAR(Write After Read)依存を軽減し性能を向上することが可能である．

スライド・ウィンドウアーキテクチャはレジスタ改名機構の一種であり，ソフトウェア・パイプラインによる擬似ベクトル処理においてメモリの速度の遅さを隠蔽する目的も兼ねて超並列計算機CP-PACSのノードプロセッサに実装されている[INBN93]．全部で128本存在する浮動小数点数レジスタのうち同時に参照できるのは32本である．

インテルのIA-64アーキテクチャ[Int99, Ike00, Jin99]に実装されたローテイティング・レジスタも改名機構の一種で，全部で128本存在する浮動小数点数レジスタをすべて同時に参照可能である．最も小さいレジスタ番号と最も大きいレジスタ番号が輪のように連続している特徴を持つ．

どちらもレジスタ改名機構によって改名されないレジスタを持っており，ループにおける定数などを格納しておくのに利用する，図2.1はローテイティング・レジスタとスライド・ウィンドウアーキテクチャの違いを示したものである．

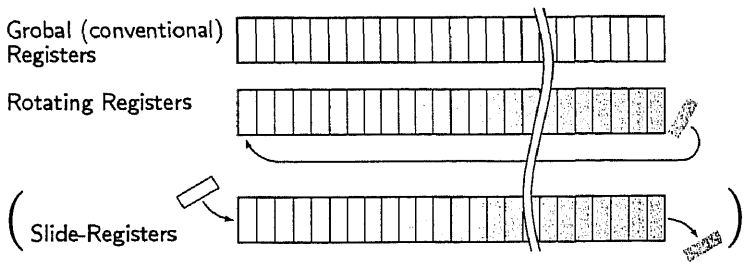

図 2.1: ローテイティング・レジスタ

2.4 干渉グラフと彩色アルゴリズム

2.4.1 レジスタ干渉グラフ

レジスタ割付方法は一般的に，グラフとアルゴリズムからなる．レジスタ干渉グラフ（Register Interference Graph）と，グラフに対する彩色アルゴリズムを用いたレジスタ割り付け方法は，現在まで最も良く利用されている手法である [CAC+81, Cha82].

レジスタ干渉グラフは，各々の変数の生存区間をグラフ上のノードとして表現し，同時に存在する生存区間を無向エッジで結んだグラフである．エッジで結ばれたノード同士は同じ実レジスタに割り付けられることはない．このとき，各々のノードに接続されている線分の数をノードの**度数**と呼ぶ．レジスタ干渉グラフの例を図 2.2 に示す.

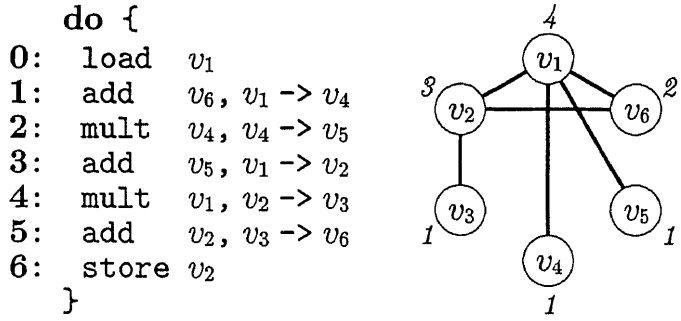

図 2.2: レジスタ干渉グラフ

レジスタ干渉グラフに対してグラフへの彩色を考える．エッジで結ばれたノードどうしを別の色で彩色すると，レジスタ番号を色に見立てることでレジスタ割り付けが行なえる.

本研究では，使用可能な実レジスタの上限を考えず，占有レジスタ数の最小化を目的とする．実際には，使用可能レジスタの数に応じて，レジスタあふれ処理などを行なう必要がある.

一般のグラフにおいて，グラフをある色数以下で彩色する問題は理論的にはNP完全な問題であり [Sas89]，彩色数が最小となる彩色結果を求めるのも同様に困難である．だがレジスタ割付の場合には，実行環境であるプロセッサのレジスタ数の制限に収まる彩色を求める条件で，最適ではないが実用的な彩色結果が得られる k 彩色法 [Sas89] のような近似アルゴリズムが存在する．k 彩色法はレジスタ割付法として現在でもよく使われている [DKK+99].

2.4.2 スライドレジスタ干渉グラフ

レジスタ干渉グラフを用いた割付方法をレジスタ改名機構の一種である，スライドウィンドウ・アーキテクチャに適用できるように拡張する方法が提案されている [HSYN98].

図2.3に示したプログラムは，図2.2のプログラムにレジスタ改名命令（$slide+1$）を付加したものである．このプログラムにおいては，v_6 は，生存区間がステップ5から（次の繰返しの）ステップ1までであるから，生存中にスライド命令を通過してレジスタ番号が変化してしまう．そのため，このようなレジスタ番号の変化を表すことのできるグラフとして図2.3に示されるスライドレジスタ干渉グラフ（Slide-Register Interference Graph）を考える．

```
    do {
0:   load  v1
1:   add   v6, v1 -> v4
2:   mult  v4, v4 -> v5
3:   add   v5, v1 -> v2
4:   mult  v1, v2 -> v3
5:   add   v2, v3 -> v6
6:   store v2
     slide +1
    }
```

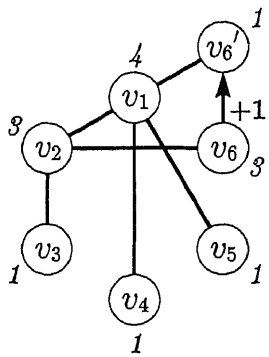

図 2.3: スライドレジスタ干渉グラフ

スライドレジスタ干渉グラフでは，元の v_6 を，スライド命令通過前後で分割して考える．具体的には通過前を v_6，通過後は通過前の名前に ' を付して v_6' とし，その間を有向線分で結ぶ．有向線分上にはレジスタ番号の変化量 S を記し，有向線分の元の頂点のレジスタ番号に S を加えると有向線分の先の頂点のレジスタ番号になるものと定める．このようにしてスライド命令による依存を表現する．

定義 2.1 （頂点の度数）
スライドレジスタ干渉グラフにおける各頂点の度数を次のように定める．

- その頂点から有向線分が出ていないとき

  頂点から出ている無向線分の数

- その頂点から有向線分が出ているとき

  頂点から出ている有向，無向線分の数と，有向線分の先にある頂点の度数の合計

2.4.3 簡単な彩色近似アルゴリズム

この種の彩色問題では，制約の大きいものから先に彩色し，制約の少ないものを最後に彩色するのが一般的である．なるべく少ないレジスタ数で彩色するためには，ごく簡単には，次のような近似アルゴリズムで実現できる．

アルゴリズム 2.1 彩色アルゴリズム *1*（*COLORING*）

1. 未彩色の頂点のうち，度数の最も大きいものを *1* つ選ぶ，無ければ終了．
2. 選ばれた頂点と無向線分で結ばれている頂点のうち，すでに彩色されているものがあれば，そのリストを作る．
3. 選ばれた頂点を，リストに含まれない最小の非負整数で彩色する．
4. *1* へすすむ．

2.4.4 k-彩色近似アルゴリズム

グラフの頂点を k 色以下の色で塗ることを，グラフの k-彩色（問題）という．最適ではないが実用的な近似アルゴリズムが与えられている [Sas89]．本稿では彩色数の最小化を考え，小さい k から順次適用し最小の k を彩色数とする．

アルゴリズム 2.2 彩色アルゴリズム *2*（k-*COLORING*）

1. 与えられたグラフ G 中で度数が $k-1$ 以下の頂点を一つ選び（それを n とする），頂点 n と n から出ている辺を取り除き，グラフ G' を得る．
2. G' を G とおいて（*1*）を繰り返すと，(*i*) 空なグラフが得られるか，(*ii*) すべての頂点の度数が k 以上であるようなグラフが残る．
3. (*i*) の場合次のように *(1)* の繰り返しを逆にたどる．G' の k-彩色に，n と n から出ていた辺を加え，n に「n から辺で結ばれている頂点と異なる色」を塗って，G の k-彩色を得る，ことを繰り返す．これで，もとのグラフの k-彩色が行える．
4. (*ii*) はレジスタが k 個では足りない場合である．

2.5 Spiral Graph と Short Bridge Algorithm

2.5.1 Spiral Graph

Spiral Graph は，ループにおける変数の生存区間表現に適した Cyclic Interval Graph[HGAM92a, HGAM92b] を，スライドウィンドウ・アーキテクチャやローテイト・レジスタのようなレジスタ改名機構を素直に表現できるように拡張したグラフである [HSYN98, Har00].

Spiral Graphのを示したのが図2.4である．図の縦軸のらせんは実レジスタに相当し，横軸はソフトウェア・パイプラインカーネル内の命令サイクルを示す．

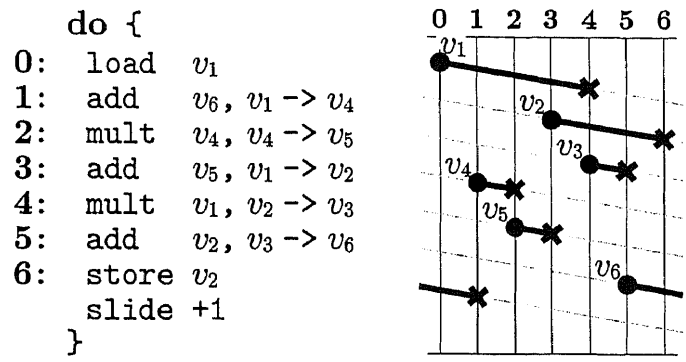

図 2.4: Spiral Graph

レジスタ干渉グラフ上にレジスタ改名機構を表現するためには，改名前のノードと改名後のノードを用意しこれらの関係を定義しなければならなかった．これに対してSpiral Graphでは，1つの実レジスタは改名を含めて1本の連続した螺旋(らせん)で表現されており，非常に単純にレジスタ改名機構を表現できる．

2.5.2 Short Bridge Algorithm

Short Bridge Algorithmは，Spiral Graphの螺旋に変数の生存区間を割付けていくアルゴリズムである．

生存区間の存在する螺旋の数と長さが，必要となるレジスタ数に相当するから，変数の生存区間と生存区間の間の隙間，すなわち図2.5における生存区間の隙間(gap)ができるだけ小さくなるように順番に生存区間を並べる方法である．

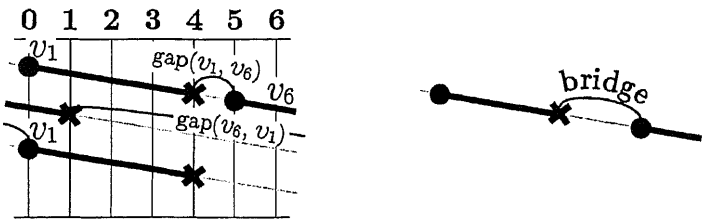

図 2.5: Short Bridge Algorithm

Short Bridge Algorithmは計算量が非常に小さいにもかかわらず，多くの場合最適解を得られる．

2.6 述語付き命令実行機構

述語付き命令実行機構は，それぞれの命令に対して**述語（predicate）**による修飾を行なっておき，述語の値が真である場合には命令を実行し，偽である場合には，命令がnop命令のように計算に影響を与えないように振舞うハードウェア機構である．Cydra5[BYA93]において提案及び実装がなされ，現在はIA-64アーキテクチャなどの最新のプロセッサに搭載されている．

述語付き命令実行機構によって，IF変換[AKPW83]を施したコードをそのまま実行可能であるため，条件分岐を含むコードであってもみかけの実行パスを単一なものとして扱える．IF変換は，条件分岐命令を含む命令列を，それぞれの命令に述語を付加した形式に変換する方法であり，条件分岐命令は述語の定義命令に置き換えられる．条件分岐命令が複数存在する場合には述語を複数定義し，1つの命令にはこれらを組み合わせた述語が付加されることになる．

分岐方向の予測が困難な，プログラム中の明示的な条件分岐によるジャンプ命令はパイプラインによる実行を乱し，性能の低下を招くことがある．述語付き命令実行機構には，条件判定とそれに続くジャンプ命令によって起こる予測ミスによるペナルティなどを減じる効果の他にも，ソフトウェア・パイプラインを含む基本ループの最適化技術を条件分岐の影響を受けずに適用できるようにするという利点を持つ．このようなハードウェアを持つ場合には，条件分岐を含むソフトウェア・パイプラインの実行性能が，ハードウェアが無い場合に比べて平均して34%ほど良いという実験結果も報告されている[WLmWH93]．

条件分岐構造内の計算量が多かったり，then節あるいはelse節に大幅に偏っていたりする場合には，結果としてnop命令ばかりが発行される場合もある．ジャンプ命令によるペナルティが小さい場合，あるいは条件分岐構造におけるコード量が相対的に大きいなどの理由で，ジャンプ命令での性能低下よりもnop命令の過剰発行による性能低下が大きい場合には，述語実行機構を持たない場合の最適化方法と組み合わせて利用することも考えられる．どのような場合において条件分岐向けハードウェアを使うべきか，あるいは従来のスケジュール方法を適用すべきかについては，検討の余地がある．

第3章　述語付き命令実行機構を持たないアーキテクチャにおけるレジスタ割付法

3.1　EMS とその改良法

3.1.1　条件分岐を含むコードに対するソフトウェア・パイプライン法

ソフトウェア・パイプライン処理を行なわせるための効果的な命令スケジュール法として，モジュロ・スケジューリングがある．

モジュロ・スケジューリングを，ループ中に条件分岐構造が含まれるプログラムにそのまま適用することは困難である．

条件分岐構造によって，実行される可能性のある実行パスがプログラム中に複数存在することになる．i 回目の条件判定の結果，実行されるコードには then 節，else 節の 2 つの可能性があり，同様に $i+1$ 回目で実行されるコードにも then 節，else 節の 2 つの可能性がある．ソフトウェア・パイプライン，特にモジュロ・スケジュールを用いた場合には，ループのそれぞれの部分をパイプラインとして重ねあわせるように命令スケジュールを行なうため，i 回目の繰り返しの then 節あるいは else 節の一部分に，$i+1$ 回目の繰り返しの then 節あるいは else 節の一部分が含まれることになる．このため，実行されうるすべての命令を，実行可能なタイミングに従って資源予約表に並べる通常のモジュロ・スケジューリングにおいては，これらのすべての組み合わせを 1 つの資源予約表にスケジュールすることはできない．

本論文で対象とするプログラムは図 3.1 で示すように，多重ループの最内部に条件分岐構造があるプログラムである．ループの実行回数は，ソフトウェア・パイプライン化を行なうことで性能が発揮できる程度に大きいとする．並列，あるいは多重構造となった条件判定命令の存在も許す．

実際のプログラムは，例えば図 3.2 のサンプルプログラムに代表されるようなプログラムである．

ループ中に条件分岐構造が含まれるプログラムに対しては，述語付き命令実行機構 [BYA93] を用いる方法がある．述語付き命令実行機構がある場合には，複数の実行パスを 1 つの実行パスの命令列とみなすことができるため，モジュロ・ス

```
L :     L1 :do i1 = l1, u1
        L2 :    do i2 = l2, u2
                   ...
        Ln :       do in = ln, un
                      stmtp
                   if ( cond )
                      then stmtt
                      else stmtf
                   end if
                      stmte
                end do
                ...
            end do
        end do
```

図 3.1: 一般化された条件分岐構造を含む多重ループ

ケジューリングによる命令スケジュールをほとんどそのまま適用することが可能になる [WLmWH93].

しかし，現在一般に普及しているほとんどのプロセッサは，述語付き命令実行機構を持たない．このようなプロセッサに対して，条件分岐を含む場合のモジュロ・スケジューリングを用いたソフトウェア・パイプライン法として以下の方法が提案されている．

- 階層的縮約（Hierarchical Reduction）[Lam88]

  条件分岐構造（if ～ then ～ else）を 1 つの大きな擬似命令とみなしてスケジュールする方法.

- Enhanced Modulo Scheduling（EMS）[WHB92]

  条件分岐構造を IF 変換で単一のパスに変換したのちにモジュロスケジュールし，逆 IF 変換によって述語付き命令実行機構の無いアーキテクチャで実行を行なえるようにする方法.

階層的縮約においては，通常のアセンブリプログラムにおける条件判定とジャンプ命令，及び then 節，else 節に含まれる命令をそのままの形で残し 1 つの疑似命令として扱う．この擬似命令の内部には，他の命令を発行できる空き命令スロットが存在するので，条件分岐構造に含まれない部分の命令を空き命令スロットで発行することでソフトウェア・パイプラインによる最適化が行なえる.

```
do i = 1, 1000
       if ( A[i] > 1.0 ) then
              C[I] = 2 * ( A[I] + 1 ) ** 2 + 3
       else
              C[I] = A[I] ** 2 + 4
       end if
end do
```

図 3.2: サンプルプログラム

階層的縮約は原理や実行モデルが単純であり，条件分岐命令の数やソフトウェア・パイプラインにおけるステージ数の影響を受けないが，1つの大きな擬似命令，すなわち条件分岐構造に含まれる命令群がソフトウェア・パイプラインによる最適化の恩恵を受けられないことで，II が長くなり実行性能の面で見劣りする場合がある．

これに対して**Enhanced Modulo Scheduling（EMS）**[WHB92]は，条件分岐構造内部の命令もソフトウェア・パイプライン化されて実行されることでループ立ち上げ間隔（II）を小さくすることが出来るため，実行性能が高いという特徴を持つ．

EMSでは，ソフトウェア・パイプラインのステージごとに，条件分岐構造の then 節あるいは else 節の順列となる複数のパイプライン・カーネルをあらかじめ準備しておく．簡単にいえば，

1. i 回目の繰り返しで実行されるのが then 節の場合と，$i+1$ 回目の繰り返しで実行されるのが then 節の場合のコード
2. i 回目の繰り返しで実行されるのが then 節の場合と，$i+1$ 回目の繰り返しで実行されるのが else 節の場合のコード
3. i 回目の繰り返しで実行されるのが else 節の場合と，$i+1$ 回目の繰り返しで実行されるのが then 節の場合のコード
4. i 回目の繰り返しで実行されるのが else 節の場合と，$i+1$ 回目の繰り返しで実行されるのが else 節の場合のコード

の4種類をそれぞれ別々のパイプライン・カーネルとしてあらかじめ用意しておくことになる．

実行時にはもとのプログラムの条件判定命令によって，もっとも適切なパイプライン・カーネルを選択しつつ実行をすすめることになる．

EMSには，ループ中に含まれる条件分岐の数，ソフトウェア・パイプラインのステージ数に応じて複数のパイプラインカーネルの大きさ（種類）が指数関数的

に増えるという問題があるものの，条件分岐を含むループのソフトウェア・パイプライン化法として，実行性能の面において非常に期待できる．

3.1.2 EMSによるスケジュール

EMSでは，条件分岐構造を含むループプログラムを以下の手順でソフトウェア・パイプライン化する．

1. 条件分岐構造を含むコードをIF変換[AKPW83]し，述語付き命令に変換する
2. 述語付き命令を通常の命令として資源予約表にモジュロ・スケジュールする．ただし，同じ空き命令スロットに，同じ実行サイクルで実行可能でかつ述語が異なる命令（相補的な述語を持つ命令）を同時にスケジュールできる．
3. スケジュールされた命令列に対して逆IF変換[WMmWHR93]を行なう．逆IF変換によって述語付き命令を含むコードを，通常の命令とジャンプ命令によるコードに変換することができる．
4. レジスタ割り付けのためにモジュロ変数展開[Lam88]を行なう．

条件分岐構造を含むプログラムの実行パスを単一にするために，まず**IF変換（条件変換）**を行なう．IF変換は条件判定命令とそれに続く条件分岐構造を，述語の設定命令と，述語を参照しつつ実行される命令列へと変換する．

モジュロ・スケジュールのための資源予約表には，同じ命令発行サイクルに相当する命令スロットに，述語の値ごとの空きを設けておく．通常のモジュロ・スケジュールと同様に，命令間のレイテンシや依存関係を考慮しながら命令を配置していく．EMSにおいては，述語が異なる複数の命令（相補的な述語を持つ命令）を同じタイミングで実行できるように配置できる．資源予約表の上は複数の命令が存在するが，実行される場合には条件によっていずれか1つの命令が発行されることになる．

条件分岐構造がソフトウェア・パイプライン化されるため，条件分岐判定すなわち述語の定義命令のあとの命令列もパイプライン・ステージで分割される．ソフトウェア・パイプラインにおいては，ステージごとの命令列を組み合わせ，重ね合わせるように実行されることから，パイプライン・ステージごとに述語の値を保持しておく必要がある．述語付き命令からなるカーネルが3ステージのパイプラインであるときには，1つの述語に対して3ステージ分にあたる3 bitの条件判定による真偽の履歴を保持し，これを参照しつつ命令を選択して実行する必要がある．

逆IF変換（逆条件変換）は，述語付き命令を含むコードを，条件分岐命令と述語の必要の無いコードへと変換する手法である．これによって，述語付き命令実行機構のためのハードウェアを持たない通常のプロセッサにおいても実行可能なコードが生成される．逆IF変換では，述語がとりうる真あるいは偽の組み合わせのすべての順列ごとに，個別のパイプライン・カーネルとなる基本ブロックを生

成する．複数のステージに対する述語の履歴に対しては，3 ステージ分にあたる 3 bit の真偽の履歴の順列組み合わせである 2^3 種類，すなわち 8 種類のカーネルとしてそれぞれ基本ブロックを生成する．基本ブロックはそれぞれがソフトウェア・パイプラインにより実行されうるコードの組み合わせになっており，ここでは，これらをパイプラインカーネル群と呼ぶことにする．

さらに，述語の定義命令をこれらの基本ブロック間の状態遷移命令に置き換える．

最後にモジュロ変数展開を行ない実行可能なコードとして再生成する．これよりレジスタ自分自身の値を上書きするような変数を除去できる．

3.1.3 改良 EMS によるアプローチ

本研究の先行研究として，EMS よりも実行効率の高い**改良 EMS** を提案し，さらに詳細に解析をすすめている [YN94, IY99].

ソフトウェア・パイプライン処理において，性能を決定する要因となるのが *II* の値であることはすでに述べた．

条件分岐を含むループプログラムに対して従来の EMS を用いて命令スケジュールを行なう場合，条件判定の結果が真となる場合と偽となる場合での最小 *II* が異なるときには *II* の大きい側を基準に資源予約表を作成し，スケジュールを行なっていた．*II* の小さい側を基準に資源予約表を作成すると，*II* の大きい側の命令をすべて配置することができなくなるからである．

しかし，このスケジュール結果をもとに実行を行なうと，*II* の大きい側を実行する場合にはプロセッサのリソースを過不足無く消費できるが，*II* の小さい側を実行する場合には，本来必要な *II* よりも長い間隔でループが立ち上がるため実行効率が落ちる．特に *II* の小さい側への分岐確率が高い場合には，実行効率の低下が問題となる．

改良 EMS はこの点に注目し，逆 IF 変換によってコードがカーネル群へと変換されることを前提とすることで，*II* の小さい側を基準にスケジュールを始め，実行効率を全く落とすことなく演算することを可能とした [YN94]．*II* の小さい側を実行する場合には小さい *II* で，*II* の大きい側を実行する場合には必要なだけ *II* を延ばすことで，すべての実行パスにおいて，必要最小限の命令のみを読み込んで，ソフトウェア・パイプラインによる実行を行なうことができることから，条件分岐向けの最適なソフトウェア・パイプライン法とも言える．

さらに，従来は考慮されていなかった複数の条件分岐に対する改良 EMS 法についても提案を行なった．

EMS 及び改良 EMS の特徴として，条件分岐構造の数や，ステージ数によってコード量が指数関数的に増大するという問題点がある．スケジューリング時に相補的な命令を同じ実行タイミングの命令スロットへと配置していくが，このときに適切な命令とペアを組むことによって，ほとんど性能を落とさずにパイプラインステージ数を削減し，コード量の増加を抑える方法を提案した [IYN99].

3.1.4 改良EMSによる命令スケジュール

図3.2に示される簡単なプログラムを改良EMSによって実際にスケジュールする.

改良EMSにおいては，資源予約表の大きさをIIの小さい側，このサンプルプログラムにおいては条件判定が偽の場合のelse節を実行する場合を基準に作成する.

このため，IIの大きい側，サンプルプログラムでは条件判定が真の場合のthen節においては命令を配置するスロットが不足することになる．改良EMSにおいては，必要な場合にのみ資源予約表を拡張し，命令スロットを挿入することでこれに対応する.

図3.3は，改良EMSによる資源予約表にスケジュールした結果を示す．資源予約表が曲がっている部分が，IIの大きさを考慮して命令スロットを挿入した部分である．EMS及び改良EMSの特徴となる述語の組み合わせ部分以外の詳細な命令スケジュール方法については，通常のモジュロ・スケジュールと同様である.

なお，図3.1における$stmt_e$に関しては，$stmt_t$及び$stmt_f$の後尾にそれぞれ別々に付加し，命令スケジューリング時の制約を軽減する.

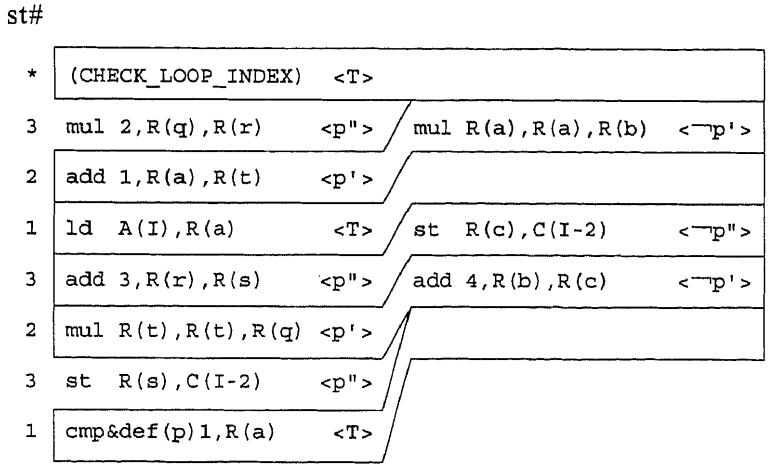

図3.3: 改良EMSにおけるモジュロ・スケジュール結果

ここで，命令はノンブロッキング型とし，必要な場合にはnop命令により命令発行タイミングを調整するものとする.

簡単のために，命令の同時発行数は1 MC（マシンサイクル）あたり1とした．整数，浮動小数点数，メモリ演算などの種類により発行命令数が異なるアーキテクチャも多いが，EMSや改良EMSにおいてこれを考慮することは，通常のモジュロ・スケジューリングでそれらを考慮することと変わらない.

命令のレイテンシは，仮想的にメモリからのロード命令（ld）及び浮動小数点数の積算命令（mul）を3MC，浮動小数点数の加算命令（add）を2MC，その他の命令を1MCと想定したものである．通常はメモリへのストア（st）にも実行

時間がかかるが，ストアバッファなどによりこれを隠蔽できると考える．また遅延分岐は考えない．

演算に使用されるレジスタは数が無限にあるものとして，R(a)，R(b) などの仮の名前を持つ**仮想レジスタ名**としてスケジュールを行なう．のちに適切なレジスタ割り付けにより**実レジスタ名**との対応をとる．

なお，1，2などは浮動小数点数の定数である．プロセッサの種類にもよるが，あらかじめ定数をレジスタにロードしておく必要がある場合もある．ここでは定数による表現をそれらのレジスタと同一視して用い，レジスタ割り付けも別に行なうものとする．

命令の右側に付した p が述語であり，CMP & DEF(p) 命令によって述語 p の値は真あるいは偽に設定される．<p>は，述語 p の値が真の場合，<¬p>は，述語 p の値が偽の場合にのみ，その命令が実行されることを示す．

<T>は，その命令が常に実行されることを示す．

p' および p'' は，ソフトウェア・パイプライン実行のために，条件判定の履歴を保持していることを示している．p' は1つ前のソフトウェア・パイプラインのステージにおける条件判定の結果を，p'' は，2つ前のステージにおける条件判定の結果を参照して，命令の実行あるいは無視を決定することを示す．

仮想命令 check-loop-index は，ループの終了判定命令である．通常はこれをカーネルの後尾に配置するのが自然であるが，それぞれのカーネルのステージごとの条件を明確にするため，あえて条件分岐命令をループの後尾に配置する方法をとる．なお，終了判定命令の位置によって実質的な違いは生じない．

この資源予約表に対するスケジュール結果を，逆IF変換して得られた，実行可能なコードが図3.4である．

命令のラベルにおける * はそのステージでの条件分岐の結果がまだ確定していないことを示す．また - はそのステージでの条件分岐の結果が存在しないことを示す．

複数の条件分岐がある場合には，述語（predicate）の数を増やす．述語の数は，条件分岐の数に応じて，述語の履歴の数は，ソフトウェア・パイプラインステージ番号に応じて増える．ラベルはこれを反映して TF/-T のように表現される．

3.2 EMS の実行モデルの解析

EMS 及び改良 EMS は，述語付き命令実行機構を持つプロセッサにおいて実行される可能性のあるすべての場合の命令列の組み合わせをそれぞれ別の命令列としてあらかじめ準備しておく方法である．実行時には，もとのプログラムの条件分岐に相当する通常の条件判定命令とジャンプ命令によって，ソフトウェア的にそれらの命令列から適切なものが選ばれる．すなわち，EMS 及び改良 EMS によって命令スケジュールされた命令列の実行モデルは，ソフトウェア・パイプライン

```
*TT: (CHECK_LOOP_INDEX)          *FT: (CHECK_LOOP_INDEX)
     mul 2,R(q),R(r)                  mul 2,R(q),R(r)
     add 1,R(a),R(t)                  mul R(a),R(a),R(b)
     ld  A(I),R(a)                    ld  A(I),R(a)
     add 3,R(r),R(s)                  add 3,R(r),R(s)
     mul R(t),R(t),R(q)               add 4,R(b),R(c)
     st  R(s),C(I-2)                  st  R(s),C(I-2)
     cmp 1,R(a)                       cmp 1,R(a)
     bgt *TT                          bgt *TF
     jmp *FT                          jmp *FF

*TF: (CHECK_LOOP_INDEX)          *FF: (CHECK_LOOP_INDEX)
     add 1,R(a),R(t)                  mul R(a),R(a),R(b)
     ld  A(I),R(a)                    ld  A(I),R(a)
     st  R(c),C(I-2)                  st  R(c),C(I-2)
     mul R(t),R(t),R(q)               add 4,R(b),R(c)
     cmp 1,R(a)                       cmp 1,R(a)
     bgt *TT                          bgt *TF
     jmp *FT                          jmp *FF
```

図 3.4: 改良 EMS における逆 IF 変換結果

化された述語付き命令列の実行状態と等価である.

図 3.3 に示した改良 EMS による命令スケジューリング結果において，条件分岐命令に支配されるのはソフトウェア・パイプラインの 3 ステージのうち，後半の 2 ステージ分である．逆 IF 変換によって，後半の 2 ステージそれぞれが then 節を実行する場合 else 節を実行する場合の命令の組み合わせを別々のコードとして生成することから，図 3.4 のような 2^2 種類すなわち 4 つのカーネルとなる.

EMS 及び改良 EMS では，図 3.4 におけるそれぞれのパイプライン・カーネルの後尾にある条件判定命令によって次に遷移すべき適切なカーネルを選択しつつ実行をすすめる．プログラムがソフトウェア・パイプライン処理によって正しく実行されるためには，ある場合に 2 ステージ目が then 節を実行していたとすると，次に遷移すべきパイプライン・カーネルは 3 ステージ目が then 節であるカーネルのうちいずれかになる．このサンプルプログラムでは，*FT は 2 ステージ目が else 節を実行している場合だから，次に遷移するのは，3 ステージ目が else 節である*TF あるいは *FF となる．逆 IF 変換されたコードにおいて，パイプライン・カーネルの制御は任意のコードから任意のコードへと遷移するわけではない．ある繰り返しにおける条件判定の結果は，パイプライン処理の終了時まで保持されるから，ある状態を示すパイプライン・カーネルからは，特定のパイプライン・カーネルへしか遷移しないことがわかる.

この状態遷移の様子を有向グラフによって表現すると，図 3.5 のようになる．逆 IF 変換によって生成されたパイプライン・カーネルの識別ラベルは，パイプライン・ステージのどの位置にあっても良いため，状態については TT や TF によって表現している．例えばサンプルプログラムにおける 1 ステージ目は条件判定命令の

影響を受けない．そのため*のステージについては図には描かないこととする．サンプルプログラムにおいて，条件判定の影響を受けるのは後半2ステージ分であるから，状態遷移図は図3.5左のようになる．それぞれのノードが，図3.4におけるそれぞれのパイプライン・カーネルに相当している．

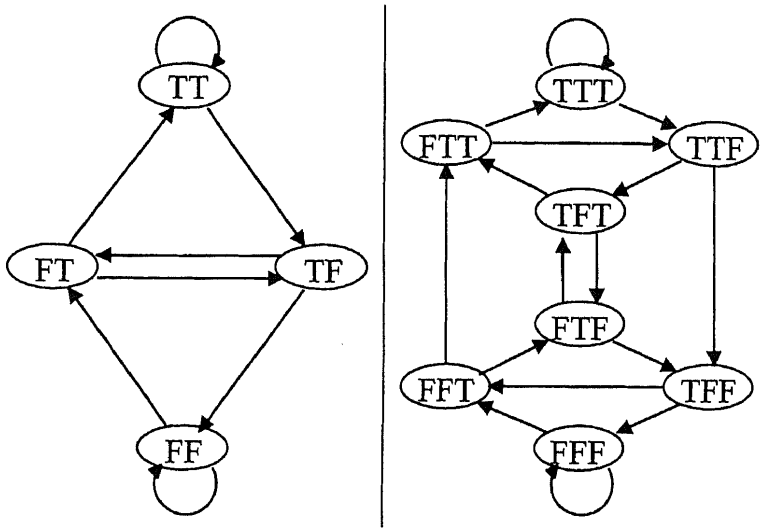

図 3.5: 状態遷移の様子

もし命令スケジュール時において，条件判定の影響を受けるパイプライン・ステージが3ステージ分存在したとすると，状態遷移図は図3.5右のようになる．

なお，この状態遷移図は擬対称な有向オイラー回路となっている．

3.3 干渉グラフを用いたレジスタ割り付け方法

EMS及び改良EMSにおいては，プログラムをIF変換によって述語付き命令に変換しスケジュールしたのち，逆IF変換によって複数のパイプライン・カーネルへと展開する．それぞれのカーネルにはステージごとに，then節あるいはelse節で全く別に使われる変数が混在する．本来はソフトウェア・パイプライン処理のための1つのループであるが，カーネル間の遷移が複雑であることから，レジスタ割り付けは従来よりも複雑になると考えられる．

本論文では，EMS及び改良EMSに対して，一般的にレジスタ割り付けによく用いられる干渉グラフを用いたレジスタ割り付け方法を提案する [IHYN98, IHYN99, IHYN02].

EMS[WHB92]の提案においては，レジスタ割り付け方法についての言及はない．使用中のレジスタへのデータの上書きなどが起こらないように，モジュロ変数展開 [Lam88] を行なうことだけが述べられている．これは，条件分岐を含むループのスケジューリングによる性能向上に重点を置いた提案だからである．

EMS 及び改良 EMS に対して干渉グラフを用いたレジスタ割り付けを行なう場合，条件判定の結果に応じて，複数の異なるパイプライン・カーネルを選択する実行モデルを考慮すると，以下の 2 つを同時に考える必要がある．

1. それぞれのパイプライン・カーネル内での生存区間どうしの干渉
2. 複数のパイプライン・カーネル間での生存区間どうしの同一性の保証

複数のパイプライン・カーネル間での生存区間の同一性の表現のために，干渉グラフに次のような記法を導入する．

異なるパイプライン・カーネルにまたがって使用されるプログラム中の変数は，同一の実レジスタに割り付ける必要がある．そこで，それらの仮想レジスタ名には ~ を付けて，ある状態を示すパイプライン・カーネル内でのみ定義され，すぐに参照される変数と区別する．さらに，同じ実レジスタに割り付ける仮想レジスタどうしを有向エッジによって結ぶことで表現する．仮想レジスタ名が $R(a\sim)$ のように，~ が仮想レジスタ名の右側に付された頂点は，頂点の存在するパイプライン・カーネルで値が定義され，別のパイプライン・カーネルで使用されることを示す．同様に $R(\sim a)$ という仮想レジスタ名は，いずれかのパイプライン・カーネルで定義された変数が，頂点の存在するパイプライン・カーネルで参照されることを表現する．

なお，すべてのステージの条件判定結果が真，すなわちすべて then 節の命令によって構成されるパイプライン・カーネルと，すべてのステージの条件判定結果が偽，同様にすべて else 節の命令によって構成されるパイプライン・カーネルの 2 つにおいてのみ，そのパイプライン・カーネル自身への状態遷移が起こる可能性がある．この 2 つのパイプライン・カーネルでは，有向エッジによって結ばれた $R(a\sim)$ と $R(\sim a)$ が同じカーネルに存在することに注意しておく．

これらをあらかじめ同一の頂点として扱い同名としてしまうことも可能であるが，ここでは拡張を考えあえて別の頂点として扱う．特にレジスタ改名機構を持つアーキテクチャにおいては，条件判定後のジャンプ命令あるいはループ制御のためのループバックジャンプの実行中にレジスタ改名処理を同時に行なうことで性能の低下を防ぐ場合がある．このような場合には $R(a\sim)$ と $R(\sim a)$ 間の有向エッジの制約として，レジスタ改名のための一定の規則を適用することが可能となる．スライドウィンドウ・アーキテクチャに対する干渉グラフを用いたレジスタ割り付け法として，スライドレジスタ干渉グラフによる方法 [HSYN98] があるが，$R(a\sim)$ と $R(\sim a)$ 間の有向エッジを，スライドレジスタ干渉グラフにおける改名の有向エッジとみなすことで，このような場合に対応できる．

3.3.1 状態別干渉グラフによるレジスタ割り付け

はじめに，干渉グラフが実際のコードの実行状態を正確に反映することを重視した状態別干渉グラフを提案する．

逆IF変換によって展開された複数のパイプライン・カーネルに制御が移ることから，まず，それぞれのパイプライン・カーネルを独立に扱い，レジスタ干渉グラフを生成する．次に異なるパイプライン・カーネル間にまたがって存在するすべての生存区間，すなわち遷移する可能性のある $R(a\sim)$ と $R(\sim a)$ の間を有向エッジで結ぶ．

図3.4に示されている，逆IF変換で複数のパイプライン・カーネルに展開されたサンプルプログラムのコードに対して，それぞれのパイプライン・カーネルごとにレジスタ干渉グラフを作成し，パイプライン・カーネル間にまたがって存在する生存区間を有向エッジで結ぶと図3.6となる．これが状態別干渉グラフの例である．

これは，従来より用いられてきたレジスタ割り付け方法を，EMS及び改良EMSにほぼそのまま適用したものであり，方法として本質的には変わるものではない．従来法との違いは，基本ブロックを越えてレジスタの同一性を表現する有向エッジを扱うことである．

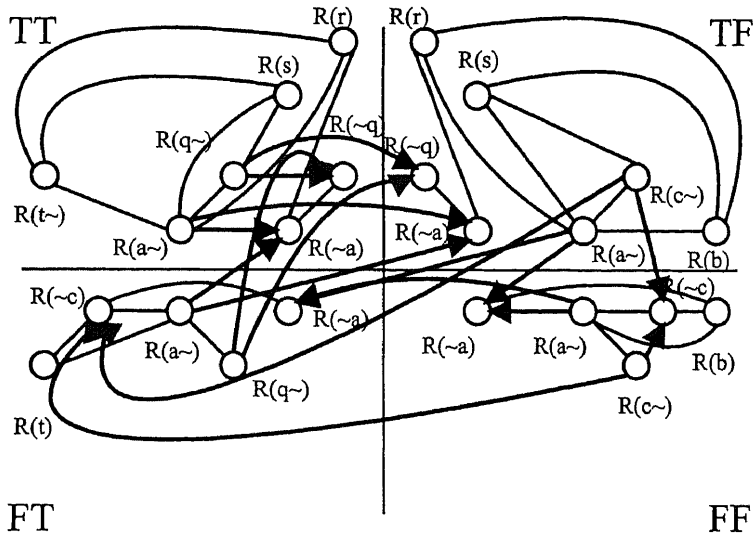

図 3.6: 状態別干渉グラフ

状態別干渉グラフに対して有向エッジにおける制約を含んだ彩色を行なう場合には次の方法が考えられる．

1. それぞれのパイプライン・カーネルに対して彩色アルゴリズムを適用し，そののちに有向エッジで結ばれた頂点が制約を満たすように色の対応を行なう．
2. それぞれのパイプライン・カーネルにおける干渉グラフどうしを，有向エッジで結ばれた頂点の制約を含めた巨大なグラフとして合体させ，これに対して彩色アルゴリズムを適用する．

レジスタ改名機構が存在しない場合，あるいは存在しても改名のための処理と

パイプライン・カーネル間の状態遷移と別に行なう場合には，有向エッジでむすばれた頂点どうしは同一色で彩色する必要がある．

前者の制約を含んだ彩色方法では，小さいグラフに対して彩色アルゴリズムを適用できるという利点があるものの，彩色後に頂点の制約が満たせない場合に，彩色アルゴリズムなどのパラメータを変更して再度彩色を行なう必要があるなどの問題点がある．後者の方法は，大きなグラフに対して彩色アルゴリズムを適用することになるが，1回の彩色アルゴリズムの適用で割り付け結果が得られる．本論文では後者を用いる．

EMS及び改良EMSによる状態遷移モデルは回路となっている[Ber70]ことから，すべてのパイプイライン・カーネルを内包した巨大なグラフに対して彩色アルゴリズムを適用することになる．

EMS及び改良EMSにおいては，パイプライン・ステージ数や条件分岐の数が増えると，逆IF変換によって展開されるパイプライン・カーネルの数が指数関数的に増える．これにともなって，干渉グラフのノード数やエッジ数についても，指数関数的に増えることが考えられる．すなわち，それぞれのパイプライン・カーネルを独立に扱い，カーネル間の変数のすべての同一性を有向エッジで表現した状態別干渉グラフでは干渉グラフ全体の大きさが巨大になり，実用的な時間での彩色が困難になる．

3.3.2 状態合併干渉グラフによるレジスタ割り付け

逆IF変換によって展開された複数のパイプライン・カーネルは，もともと1つの述語付き命令としてスケジュールされたものである．複数のパイプライン・カーネルどうしを観察すると，ステージごとの条件判定の真偽によって，共通となる命令列が必ず含まれていることがわかる．これは，干渉グラフにおいて共通な構造が必ず存在していることを示す．たとえば図3.4の*FFと*TFは，ともに第3ステージの条件が偽であることから$R(\sim c)$という全く同じ生存区間を持つ変数を含む．同一の命令列から導かれる生存区間は，干渉グラフにおける同一の構造の部分グラフとなる．

この干渉グラフの部分的な類似性に着目し，それぞれのパイプライン・カーネルごとの干渉グラフをあらかじめ重ねあわせ，見かけの大きさを縮小することで効率良くレジスタ割り付けを行なう方法として，状態合併干渉グラフを提案する．

状態合併干渉グラフは，すべてのパイプライン・カーネルにおける同名の仮想レジスタ名をすべて同一の頂点とみなしたグラフである．図3.6の状態別干渉グラフにおいて同名の仮想レジスタをあらわす頂点を同一の頂点として重ね合わせた状態として，サンプルプログラムから生成したグラフが，図3.7であらわされる状態合併干渉グラフである．

このグラフでは，必要なレジスタの干渉や同一性の表現を失わずに，グラフの複雑さを減少させて彩色アルゴリズムを適用することができる．図3.6に示された

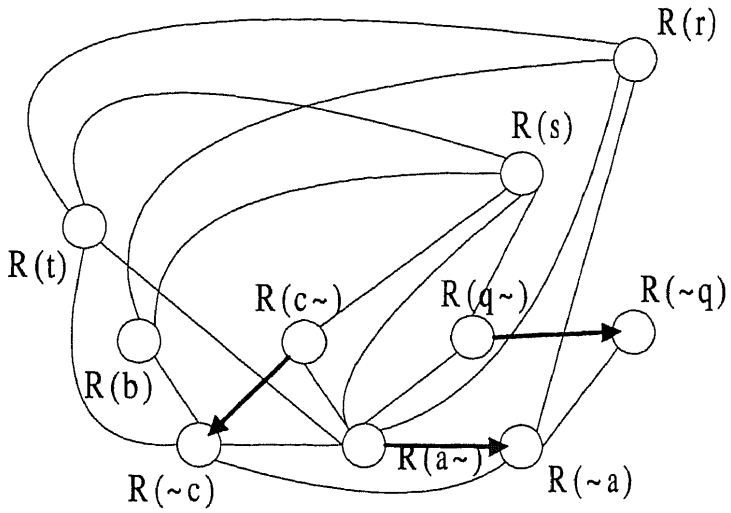

図 3.7: 状態合併干渉グラフ

状態別干渉グラフと比較して，図3.7に示される状態合併干渉グラフは，明らかに頂点や辺の絶対数が減っていることがわかる．さらに，EMS 及び改良 EMS で問題となる，パイプライン・カーネル数，すなわちコード量の増大の影響を受けずにレジスタ割り付けが可能である．

状態合併干渉グラフは，逆IF 変換による展開後の様子を示す状態別干渉グラフからではなく，展開を行なう前の状態からグラフを作成可能である．1つのパイプライン・ステージにおけるコードは，条件分岐が真あるいは偽のどちらかにのみ分岐するという排他性により，ある時点では then 節あるいは else 節のどちらかの状態しかとらない．すなわち，あるパイプライン・ステージが then 節の命令を実行しているとき，同じパイプライン・ステージの else 節の命令は実行されないことになる．ソフトウェア・パイプラインにおいては，この場合を除いて任意のパイプライン・ステージどうしを組み合わせてパイプライン処理を行なうことから，異なるパイプライン・ステージに存在する述語付きの変数あるいは述語付きの一時変数どうしは述語の値が異なっていたとしても常に干渉を考える必要がある．

よって以下の2つの条件を同時に満たす仮想レジスタどうしは，生存範囲が重なっていたとしても干渉を生じない．

1. 同一のパイプライン・ステージに存在する．
2. 互いの述語の条件が異なる．

この条件のもとで全体の干渉グラフを作成すると，状態別干渉グラフが得られることになる．この条件は，EMS 及び改良 EMS の実行モデルの解析及び状態別干渉グラフにおけるグラフから得られた．

3.3.3 グラフの重ね合わせによる問題点

EMS 及び改良 EMS においてはそのコードの性質から，干渉グラフを重ね合わせグラフの大きさを縮小した状態合併干渉グラフとすることで，状態別干渉グラフよりも効率良くレジスタ割り付けが行なえることについて述べたが，実際にはグラフの重ね合わせによって不利益が生じる場合があることが判明している．以下これを不利益条件と呼ぶことにする．

レジスタ改名機構の無いアーキテクチャや，状態遷移と同時にレジスタ改名を行なわない命令スケジュールでは，状態をあらわす複数のパイプライン・カーネルにまたがって存在する生存区間，つまり $R(a\sim)$ と $R(\sim a)$ を同一のレジスタに割り付ける必要がある．さらに，ソフトウェア・パイプラインのおけるすべてのパイプライン・ステージにおける条件判定の結果が

1. すべて真
2. すべて偽

の 2 つの場合には，そのパイプライン・カーネル自身への状態遷移が起こる可能性がある．図 3.5 左における TT あるいは FF，図 3.5 右における TTT あるいは FFF の自身のノードにかえる有向エッジがそれである．

グラフの一部を取り出して説明すると，図 3.8 左で示すような干渉が存在することが考えられる．図 3.8 左は状態別干渉グラフで表現した場合であるが，グラフにおける干渉とノードの同一性を考慮して，$R(c\sim)$ と $R(\sim c)$ を同一色に彩色したとしても，すべてのノードを 2 色で彩色できることは明らかである．よって，この部分に限っては変数をレジスタ 2 個に格納することが可能である．

この状態別干渉グラフを，同一の仮想レジスタ名すなわちノード名を同一として重ね合わせて状態合併干渉グラフの一部としたのが，図 3.8 右で示したグラフである．この場合においても $R(c\sim)$ と $R(\sim c)$ を同じ色に彩色する場合を考えると，3 頂点からなる完全グラフとして扱うことが必要となり，彩色に 3 色が必要となってしまうことがわかる．

これは，本来 1 つのパイプライン・カーネルにおける干渉が他のパイプライン・カーネルにおける同名のノードに影響を及ぼした結果であり，この例においては，状態 TT をあらわすパイプライン・カーネルの R(s) と R(t) の干渉が，他のパイプライン・カーネルでも干渉として扱われてしまったためである．

逆 IF 変換とその状態遷移の解析結果から典型的な干渉グラフを多数列挙して解析した結果，現在この場合以外に不利益が生じる場合は発見されていない．この不利益は，この例における R(s) や R(t) のように他のカーネルとは無関係に割り付けが可能な仮想レジスタが，重ね合わせによって他のカーネルの影響を受けてしまうこと，そして，すべてのパイプライン・ステージの条件判定の結果が真あるいは偽である場合に，そのパイプライン・カーネルからそれ自身の状態に制御が移る可能性があること，の 2 つの条件が組み合わされて起こると考えられる．

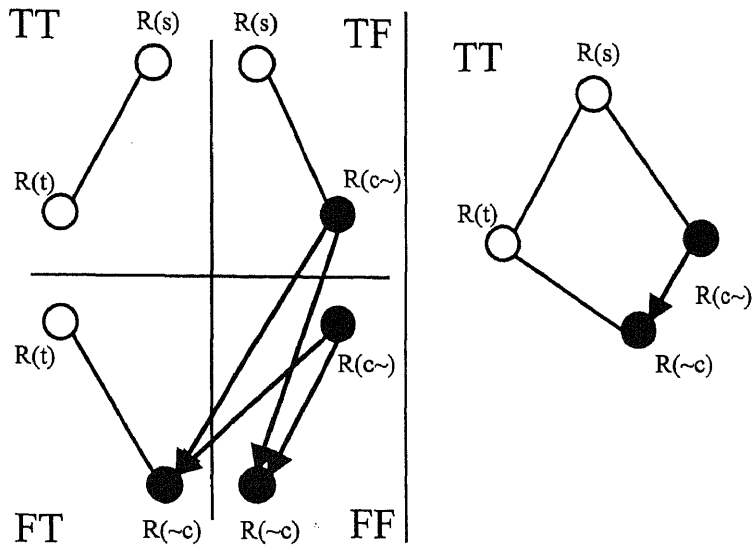

図 3.8: 重ね合わせによって色数が増えてしまう場合

この不利益の解決のために，他のカーネルとは無関係に割り付けが可能な仮想レジスタをすべて別のノードとして表現する方法もある．この方法では，状態別干渉グラフと同様にグラフが大きくなるという問題がある．そこで，このような干渉を起こしうる仮想レジスタのみを正確に発見する必要があるが，これについては現在も検討を行なっている．

なお，このような不利益条件が成立する部分が，干渉グラフ全体に複数存在したとしても，図3.8右のようなグラフから派生した完全グラフどうしが結び付くことはありえないため，グラフ全体の彩色数としてはたかだか1色増えるにとどまる．

不利益条件の起こる場合が非常にまれであることやグラフ上考えられる彩色数の増大は1色にとどまることを，近似彩色アルゴリズムの最適性と比較した場合には，大きく複雑なグラフに対して彩色アルゴリズムを適用するよりも，グラフを縮小した方が彩色に有利になると考えられる．

さらに，スライドウィンドウ・アーキテクチャにおけるレジスタ改名のための命令は，ジャンプ命令と同時に利用されることで有利に動作するため，命令スケジュールにおいてジャンプ命令と組み合わせる方針がとられている．仮想レジスタの同一性を保証するための有向グラフ部分においてスライドレジスタ干渉グラフによる拡張 [HSYN98] と同等の制約条件を設けて彩色を行なうことで，状態合併干渉グラフにおける不利益条件を生じさせる $R(c\sim)$ と $R(\sim c)$ を別の色に彩色することが可能であり，不利益条件を隠蔽することが可能である．

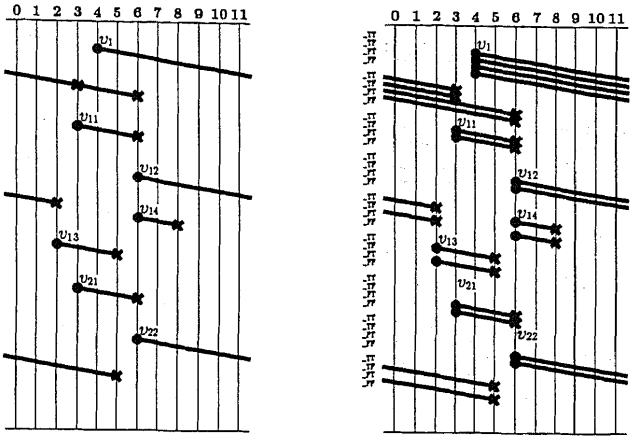

図 3.9: 改良 EMS のために拡張された Spiral Graph

3.4 Spiral Graphを用いたレジスタ割り付け方法

レジスタ改名機構を持つアーキテクチャにおいて EMS 及び改良 EMS を用した場合の干渉グラフをによるレジスタ割り付け方法としては，状態別干渉グラフ及び状態合併干渉グラフに対してスライドレジスタ干渉グラフと同様の拡張を行なう方法があることについて述べた．

これに対して，スライドウィンドウ・アーキテクチャにおけるレジスタ改名機構を素直に表現可能な Spiral Graph においても，EMS 及び改良 EMS に対する拡張方法が提案されている [Har00]．

EMS 及び改良 EMS では，逆 IF 変換によって複数のパイプライン・カーネルを別々に生成し，それらの間の状態遷移を表現する必要がある．Spiral Graph における拡張法の提案では，この別々のカーネルを表現するために，実レジスタを表現するトラックを複数とし，それぞれ別のコードに対するトラックとして扱う．さらに，状態遷移が起こるコードの後尾への生存区間の割り付け時に，状態遷移図から求められるトラック間の関係を保つようにする．述語によって修飾された図 3.9 左における生存区間は，実際には図 3.9 右のように，それぞれの別のパイプライン・カーネルでの実レジスタを占有するため，別のらせんで表現される．この拡張により，ある程度適切な割り付けが行なえることが実験によって判明したと述べられている．

しかし，Spiral Graph は単一の基本ブロックループに対してレジスタ改名機構を表現する方法として定義されているため，1 つの Spiral Graph で複数のトラックの接続性を保証する必要があり，この条件が複雑なものとなっている．例えば図 3.9 右の左端と右端で状態遷移が起こることで，生存区間 v_{12} の複数トラック間の遷移の関係を，Spiral Graph とは別の制約によって規定する必要がある．このように，複数のパイプライン・カーネル間の状態遷移を素直に表現するのが困難

なSpiral Graphを，EMS及び改良EMSに利用するのは適当ではないと考える．

本論文の第4章で述べるように，Spiral Graph単一の基本ブロックループへ適用した際に非常に有効であることから，述語付き命令実行機構によって単一の実行パスへと変換した条件分岐構造に対してSpiral Graphを用いるべきであると考える．

3.5 レジスタ割り付け実験

3.5.1 実験の目的

EMS及び改良EMSに対するレジスタ割り付け方法として，状態別干渉グラフと状態合併干渉グラフを提案した．状態別干渉グラフにおいてはグラフの巨大化による彩色の困難さ，状態合併干渉グラフにおいてはグラフの不利益条件による彩色結果の悪化の可能性の問題がある．

そこで，それぞれのグラフの正確性及びノード数減少による近似彩色アルゴリズムとの関係を調査するために，レジスタ割り付け実験を行なった．

3.5.2 実験方法

改良EMSを実装したコンパイラが存在しないため，改良EMSによって命令スケジュールされたコードを想定して，

1. 標準的なベンチマークの1つであるLivermore Fortran Kernel[McM86]から，16番ループの前半部分をハンドコンパイルしたもの
2. 乱数によって一般的なプログラムにみられるデータ依存グラフを自動生成したもの

の2種類から生存区間を抽出し，同じ生存区間群に対して，状態別干渉グラフと状態合併干渉グラフを用いたレジスタ割り付けを行なった．近似彩色アルゴリズムとして，

1. 度数順に彩色していく単純な彩色アルゴリズム（COLORING → 第2.4.3節）
2. 度数と逆順にノードを刈っていくk-彩色の応用アルゴリズム（k-COLORING → 第2.4.4節）

の2つを用いた．

なお，実験には筑波大学計算物理学研究センターのIBM RS/6000 （Power2 160MHz） を用いた．

3.5.3 ベンチマークによる評価結果

図3.10に示されるLivermore Fortran Kernel内の16番ループ（Kernel #16）の前半部分に対して改良EMSによるスケジュールを行なった．得られた変数の生存区間に対して，第2.4.3節の彩色アルゴリズム1（COLORING）及び第2.4.4節の彩色アルゴリズム2（k-COLORING）による彩色を行なった．

```
DO K=1,N
 K2=K2+1
 J4=J2+K+K
 J5=ZONE(J4)
 IF ( J5<N )
  IF ( J5+LB<N )
   TMP(K)=PLAN(J5)-T
  ELSE
   IF ( J5+II<N )
    TMP(K)=PLAN(J5)-S
   ELSE
    TMP(K)=PLAN(J5)-R
   END IF
  END IF
 ELSE
  IF ( J5=N )
   EXIT DO-LOOP
  ELSE
   K3=K3+1
   TMP(K)=D(J5)
       -(D(J5-1)*(T-D(J5-2))**2
       +(S-D(J5-3))**2+(R-D(J5-4))**2
  END IF
 END IF
END DO
```

図 3.10: Livermore Fortran Kernel #16.

スケジュールで仮定したプロセッサの条件は，第3.1.4節で述べた条件と同一である．このプログラムには複数の条件分岐が存在するため，逆IF変換によって12の部分からなる8種類のカーネルとして展開された．

彩色結果を表3.1に示す．

Livermore Fortran Kernel #16に対するレジスタ割り付け実験から，状態別干渉グラフと状態合併干渉グラフにおいては彩色数についてはほとんど変わらないものの，彩色アルゴリズム2（k-COLORING）において状態合併干渉グラフのほうがわずかに良い結果を与えていることがわかる．これは干渉グラフにおいて彩色の困難さの指標となる干渉グラフのノード数が，状態別干渉グラフでの156ノードに対して，状態合併干渉グラフとしてノードを重ね合わせることで34ノードに

表 3.1: LFK #16に対する彩色結果

	coloring	k-coloring	ノード数
	単位:レジスタ数（個）		単位:個
状態別干渉グラフ	10	13	156
状態合併干渉グラフ	10	12	34

減少していることによると考えられる.

3.5.4 自動生成プログラムによる評価結果

改良EMSによって命令スケジュールされたコード想定した，乱数による変数の生存区間群の生成プログラムを作成した．これにより生成された例題は，変数の生存区間である仮想レジスタの定義と使用の時刻を並べたもので，途中に条件分岐を1つ含む．すなわち述語によって，条件判定命令前の生存区間群，条件が真であった場合のthen節における生存区間群，条件が偽であった場合のelse節における生存区間群の3種類に区別される．通常のプロセッサにおいては，浮動小数点数の加算や乗算などにより2つの値から1つの演算結果を得られるから，このようなデータの依存性を考慮し，生存区間群も木状となるように工夫されている.

例題の仮想レジスタは各々の生存区間が1MC（マシンサイクル）から2MC，1MCから3MC，1MCから4MCまでのそれぞれ1000例，合計3000例である．それぞれの例において，IIは8から15のある値をとり，仮想レジスタとしての生存区間を15から40個含むこととした．なお命令の同時発行数は最大3命令を仮定してスケジュールした．生存区間の長さが様々なのは，主記憶からの値のロード命令のレイテンシが長くなった場合にどのような影響があるかを調査するためでもある.

これらの生存区間生成プログラムによって生成された例題から，状態別干渉グラフと状態合併干渉グラフを作成し，第2.4.3節と2.4.4節で述べた近似彩色アルゴリズムを用いてグラフの彩色を行なった．状態別干渉グラフにおいてはグラフが巨大になりすぎる例が存在したため，近似彩色アルゴリズムの適用時間に制限を設けた．実用的な時間をはるかに超える最長300 CPU秒の時間制限を超えても彩色が終了しなかったものは，彩色結果が得られなかったものとした.

一般に知られているk-彩色アルゴリズムは，レジスタ数の上限があらかじめ決められているときに，彩色が可能であるかを求める彩色アルゴリズムであるが，本論文における彩色アルゴリズム2（k-COLORING）は，k値を1から増える方向に増大させつつ順次適用することで，可能な限り小さい彩色数を求める方法として用いていることに注意する.

実験結果として得られる彩色数の評価には，状態別干渉グラフ及び状態合併干渉グラフに対して，最適な彩色が得られているかの判定が困難であるため，状態

表 3.2: 彩色アルゴリズム 1（coloring）による彩色数の差

状態合併干渉グラフ彩色数 − 状態別干渉グラフ彩色数	生存区間の長さ 単位:MC 1–2	1–3	1–4
−2	0	2	1
−1	44	107	130
±0	855	730	707
+1	101	159	157
+2	0	2	5
+3	0	0	0
（no ans.）	（0）	（0）	（19）

表 3.3: 彩色アルゴリズム 1（coloring）で彩色に要した CPU 時間の平均

	生存区間の長さ 単位:MC 1–2	1–3	1–4
状態別干渉グラフ	217	640	2024
状態合併干渉グラフ	19	20	19

単位:ミリ CPU 秒

別干渉グラフを基準として，状態合併干渉グラフの彩色数を相対的に評価する方法をとった．

彩色アルゴリズム 1（COLORING）による彩色結果の比較を表 3.2 に示す．−2（+2）は，基準となる状態別干渉グラフに対して，状態合併干渉グラフが 2 色少ない（多い）彩色を与えたことを表わす．表 3.2 における 19 例（no ans.）は，グラフが巨大であるために，状態別干渉グラフにおいて CPU 時間の制限内彩色を行なうことができなかった例題の数である．

表 3.3 は 1000 例の彩色に要した CPU 時間の平均である．

状態別干渉グラフと状態合併干渉グラフの頂点数の平均を比較すると表 3.4 のようになる．同一の例題に対しても数倍の大きさになっていることがわかる．

彩色アルゴリズム 2（k-COLORING）による彩色結果の比較を表 3.5 に示す．彩色アルゴリズム 1（COLORING）と同様に，−2（+2）は，状態別干渉グラフを基準として状態合併干渉グラフがそれよりも 2 色少ない（多い）彩色を与えたこと

表 3.4: グラフの頂点数の比較

	生存区間の長さ		
			単位:MC
	1–2	1–3	1–4
状態別干渉グラフ	100.17	165.02	278.56
状態合併干渉グラフ	27.38	28.58	30.76

表 3.5: 彩色アルゴリズム 2（k-coloring）による彩色数の差

状態合併干渉グラフ彩色数 − 状態別干渉グラフ彩色数	生存区間の長さ		
			単位:MC
	1–2	1–3	1–4
−2	0	0	0
−1	0	0	0
±0	726	488	376
+1	271	479	499
+2	3	33	114
+3	0	0	11
(no ans.)	(0)	(0)	(0)

を表わす.

表 3.6 は 1000 例の彩色に要した CPU 時間の平均である.

彩色アルゴリズム 1（COLORING）を用いた場合には，自動生成された 3000 例の例題において，7 割から 8 割の例題で状態別干渉グラフと状態合併干渉グラフに対する彩色で同じ結果が得られた．正確なグラフを表現できる状態別干渉グラフに対して，グラフの正確さでは有利にはなりえない状態合併干渉グラフを用いた場合に，1 割程度の例題で彩色数が改善されていることがわかる.

これに対して，彩色アルゴリズム 2（k-COLORING）を用いた場合には，ほぼすべての例題で状態合併干渉グラフとして重ね合わせた場合に彩色数が増えてしまっており，重ね合わせによって問題が起きているように見える.

そこで，状態別干渉グラフと状態合併干渉グラフを用いたそれぞれの場合について，彩色アルゴリズム 1（COLORING）と彩色アルゴリズム 2（k-COLORING）での彩色結果を比較すると表 3.7 となる．この表における −2（+2）は彩色アルゴリズム 2（k-COLORING）を基準に，彩色アルゴリズム 1（COLORING）が 2 色

表 3.6: 彩色アルゴリズム 2（k-coloring）で彩色に要した CPU 時間の平均

	生存区間の長さ 単位:MC		
	1–2	1–3	1–4
状態別干渉グラフ	375	3136	16302
状態合併干渉グラフ	15	18	22

単位:ミリ CPU 秒

表 3.7: 彩色アルゴリズムどうしの比較

coloring 彩色数 − k-coloring 彩色数	状態別干渉グラフ 含まれる生存区間の長さ 単位:MC			状態合併干渉グラフ		
	1–2	1–3	1–4	1–2	1–3	1–4
−3	0	0	0	0	0	3
−2	0	0	0	2	12	35
−1	0	2	19	219	270	338
±0	911	688	571	687	596	516
+1	89	298	376	92	122	84
+2	0	12	15	0	0	5
(no ans.)	(0)	(0)	(19)	(0)	(0)	(0)

少ない（多い）彩色を与えたことを表現している．

彩色アルゴリズム 2（k-COLORING）は同じグラフに対しても，彩色アルゴリズム 1（COLORING）よりも彩色数が多い結果を与えることが多い．これは，状態別干渉グラフに対して彩色アルゴリズム 2（k-COLORING）が有利に彩色を行なえるのに対して，状態合併干渉グラフに対しての彩色アルゴリズム 2（k-COLORING）が非常に悪い彩色結果を与えていることによる．

さらに，表 3.3 及び表 3.6 の彩色に必要な時間で示されるように，彩色処理が 100 倍から 1000 倍も高速化されている．

例題に含まれる生存区間が長くなるほど，彩色に必要な計算時間が増え，彩色が困難になっていることがわかる．これは 1 つの変数の生存区間が長くなることで，他の変数との干渉が増えることや，例題全体の時間が長くなることで EMS 及び改良 EMS によるスケジュール結果のパイプライン・ステージ数が増え，逆 IF 変換によるパイプラン・カーネル数が増大することによる．

3.6 レジスタ割り付け実験の考察

常に正確なグラフを扱える状態別干渉グラフに対して，状態合併干渉グラフは不利益が生じる場合を考慮してもレジスタ割り付けにおける彩色結果に遜色はない．さらに状態合併干渉グラフではEMS及び改良EMSのコードの展開の影響が小さいことから頂点数が少なく，短かい時間で彩色が行なえる．これは，複数の彩色アルゴリズムを適用することばかりではなく，不利益条件の検出を行なうことにも有利であると考えられる．

状態別干渉グラフにおける19例（表3.2のno ans.）に至ってはグラフが巨大であるため，CPU時間の制限内に彩色を行なうことができなかった．このような例題に対しても，グラフの重ね合わせによる縮小を行なうことで，ごく短い時間で実用的な彩色が可能となる．

状態合併干渉グラフにおける不利益条件が，その性質上IIが命令のレイテンシに対して非常に小さい場合に起きがちであること，そして，実際の近似アルゴリズムによる彩色を行なったとき，状態別干渉グラフが大きいことによる彩色の困難さや彩色の非最適性が，彩色数により大きな影響を与えたことなどから，多くの場合において，グラフを重ね合わせた状態合併干渉グラフにより利点があると考えられる．

また，この実験結果においてはk彩色法の連続適用を用いて彩色数を求めた場合の結果が良くないことが示されているが，グラフの枝の存在確率[Man85]が大きい場合に，単純なk彩色法での彩色結果が悪いことが原因だと考えられる．この場合には，他の近似アルゴリズムを組み合わせて用いることで彩色結果の改良が可能なことが判明している[IY99]．状態合併干渉グラフには，様々な特徴を持つ複数の彩色アルゴリズムを高速に適用することでさらなる最適化が可能であるという利点も存在する．

さらに状態合併干渉グラフにおける不利益条件が，その発生の仕組みから，レジシタ改名機構を持つアーキテクチャにおいては生じにくいと考えられるため，レジスタ改名機構を持つアーキテクチャへの干渉グラフを用いたレジスタ割り付け方法についてさらに検討中である．

3.7 関連研究

ソフトウェア・パイプラインに関するレジスタ割り付けの研究には，命令スケジュール方法も含めて大きく分けて2つの方向がある．

1. レジスタについての制約は考えずに命令スケジューリングを行ない，命令スケジューリングののちにレジスタ割り付けを行なう方法（prepass scheduling）
2. レジスタにおける制約に留意しつつ，あるいはレジスタ割り付けをあらかじめある程度行なったのちに，レジスタ割り付けと命令スケジューリングを行

なう方法（postpass scheduling）

ソフトウェア・パイプラインのようなループ最適化においては，命令レベルの並列性を出来る限り抽出するために，レジスタの制約を考えずに命令スケジュールを行ない，スケジュール済みの命令列に対してレジスタ割り付けを行なう方法が多く用いられている [CLM+95]．本研究においてもこの方針を用いている．レジスタ割り付け結果を再度命令スケジューラに対してフィードバックすることで，さらに良い命令スケジューリング結果を得られる可能性もある．

モジュロ・スケジューリングに代表される命令スケジュール結果に対してレジスタ割り付けを行なう方法は，通常のループに対するレジスタ割り付けと同様であるため，干渉グラフを用いた方法 [CAC+81, Cha82] や Cyclic Interval Graph[HGAM92a, HGAM92b] を用いた方法がそのまま適用されることが多い．Meeting Graph を用いた方法 [ELM95] は，レジスタ数を最小化するための数学的な定義を行なったものである．

ソフトウェア・パイプラインの命令スケジューリングにおいては，より高性能なスケジュール結果を得ることが重要であったため，スケジューリング法の提案においても，レジスタ割り付けを重視していなかった．Enhanced Modulo Scheduling（EMS）[WHB92] の提案においても，モジュロ変数展開 [Lam88] によりパイプライン処理によるレジスタの値の上書きを防ぐことで十分であるとし，実際のレジスタ割り付け手法については重視していない．改良 EMS[YN94] ではレジスタ割り付けを適切に行なわなければ命令スケジュールが不能になるという言及があるが，改良 EMS における実際のレジスタ割り付けにおける問題点はわかっていなかった．

干渉グラフを用いたレジスタ割り付け方法への，述語付き命令実行機構への拡張方法が提案されている．述語が異なる仮想レジスタどうしは，同時に同じ実レジスタに割り付けることが可能であるから，これらをあらかじめ束ね（bundle）て干渉グラフにおいて 1 つのノードとし，レジスタ割り付けを行なう方法が提案されている [ED95]．しかしながら述語の履歴を参照しつつ全く異なるコードを選択しつつ実行する EMS 及び改良 EMS にこの方法を用いると，状態別干渉グラフで得られる正確な干渉グラフが得られない点や，複数のパイプラインカーネルにまたがる仮想レジスタを扱うことが難しいことから十分なアルゴリズムとは言えない．さらにこの方法において比較の下限として利用している方法は，ステージ数が 1 の場合における状態合併干渉グラフと類似したグラフを与えることから，状態合併干渉グラフよりも精度の点で問題点があるとも考えられる．

レジスタ割り付けをあらかじめ行なったのちに命令スケジューリングを行なう方法もある [HG83, BEH91, Pin93, BSBC95, CS99]．これはプロセッサの持つレジスタが乏しい場合や，特定用途向けのレジスタを利用したい場合に有効である．レジスタ数を少なく抑えられることなどの利点もあるが，性能は上がりにくい．また，スケジューリング前の正確な必要レジスタ見積もりが困難であるという問題点がある．さらに，あらかじめ必要レジスタ数を見積っておくことで，ソフトウェア・パ

イプラインの深さを決定するヒントとして使う方法も考えられる．命令スケジューリングと同時にレジスタ割り付けをしていく方法も存在するが，レジスタをあらかじめ指定しておくことは命令スケジューリングの制約となるため，命令スケジューリング法の一種あるいは拡張として提案されている [WKEE94, EDA, DG98].

第4章　述語付き命令実行機構を持つアーキテクチャにおけるレジスタ割り付け法

4.1　干渉グラフを用いたレジスタ割り付け方法

第2.6節で述べたように，**述語付き命令実行（Predicated Execution）機構**を持つアーキテクチャが科学技術計算用のスーパーコンピュータ Cydra5 向けに提案，実装された [BYA93]．さらに現在では，一般向けのパーソナルコンピュータ用プロセッサである IA-64 アーキテクチャ [Int99, Ike00, Jin99] においても，ソフトウェア・パイプライン実行補助機構として述語付き命令実行機構が導入されている．

述語付き命令実行機構を持つ場合のコードに対する，干渉グラフと彩色アルゴリズムを用いた方法が提案されている [ED95]．

この方法では，命令をスケジューリングしたのちに，通常の干渉グラフを用いた方法と同様に，プログラム中の変数あるいは演算のための一時変数の生存区間をノードとして表現し，コードの実行時に同時に存在するノード間を無向エッジで結ぶことで干渉グラフを作成する．さらに，条件分岐によって，条件判定が真の場合にしか実行されないコードに存在するノードと，条件判定が偽の場合にしか実行されないコードに存在するノード間に存在するエッジを除去し，それらのノードを束ねる（バンドルする）ことによって新たな1つのノードを生成する．新たなノードは，干渉グラフ上は通常のノードと変わらないが，条件判定が真の場合と条件判定が偽の場合において，同じレジスタに対して異なる値を格納するという意味をもつ．

ノードを束ねる際に，まず3つのパラメータによってバンドル候補順を決定し，次にバンドル候補順に並んだノードからどれを選んで束ねるかの方針を4つ挙げてこれらを比較している．バンドル候補順は生存区間の始点順，生存区間の長さ順，生存区間の干渉枝数順の3種類であり，どれを選んで実際に束ねるかは，候補順の先頭にあるもの，候補順の後尾にあるもの，生存区間の重なりが大きいもの，そして生存区間の重なりと他のノードとの干渉状況から得られる関係式を最小にするものの4つである．

干渉グラフを用いたレジスタ割り付け方法としては妥当ではあるが，ヒューリ

スティック法によってノードどうしを束ねることによって新たなノードを生成し，このグラフに対して彩色を行なう方法には問題がある．もとの複数のノードと，生成されたノードが最適な組み合わせかつ完全に一致していない限り，彩色アルゴリズムが最適な解を与えたとしても，グラフから最適な割り付けが得られている保証がないからである．

また，この研究においてはソフトウェア・パイプラインのパイプラインステージが異なるノードどうしの干渉への対応方法が述べられていない．さらに，レジスタ改名機構に対する拡張方法については全く考慮されておらず，スライドレジスタ干渉グラフ（第2.4.2節）と同様の拡張を行なうことを考えると第3章で述べたようにノード数や干渉エッジが増えることで彩色が困難になることに加えて，レジスタ改名を表わす有向エッジ間の制約が存在することで，ノードどうしを束ねることが困難になることも考えられる．

変数どうしの干渉のみを扱う干渉グラフよりも，生存区間の始点，終点の情報を用いた方法は多い情報からレジスタの割り付けを決定できるためより良い結果が得られると予想できる．述語付き命令実行機構により実行パスを単一に変換できることで，生存区間を用いたレジスタ割り付け方法が適用しやすくなった．さらにレジスタ改名機構を持つ場合には，Spiral Graph[HSYN98]のような素直なグラフを用いることで，効果的なレジスタ割り付けが期待できる．

4.2 Spiral Graphを用いた方法

4.2.1 レジスタ改名機構と述語付き命令実行機構をあわせもつアーキテクチャ

パーソナルコンピュータ向けに普及しはじめているIA-64アーキテクチャにおいては，ソフトウェア・パイプライン処理によるコード実行を補助する目的で，いくつかのハードウェアを用意している．

1. プレディケーション（述語付き命令実行機構）
   IA-64アーキテクチャが備える64本のプレディケートレジスタ（p0 ～ p63）を使い，命令の述語としてすべての命令にプレディケートレジスタ番号を指定する．
   すべての命令は，1bitのプレディケートレジスタの値を参照し，これが1であった場合には命令が実行され，0であった場合にはnop命令と同様に実行結果が反映されない．
   なお，p0は常に1になっており，プレディケーションを利用しない場合にはこのプレディケートレジスタ番号を命令に付加することになっている．
2. ローテイティング・レジスタ（レジスタ改名機構）
   IA-64アーキテクチャが備える128本の64bitの汎用レジスタ（r0 ～ r127）

及び浮動小数点数レジスタ（f0 ～ f127）のうち，r32～ r127 及び f32 ～ f127 の 96 本ずつを，一斉にレジスタ名が変更可能なレジスタ改名機構付きレジスタ群として利用することが可能である．
さらにプレディケートレジスタ p16 ～ p63 の 48 本は，汎用レジスタ，浮動小数点数レジスタと同様にレジスタ改名機構付きレジスタ群として利用することが可能である．

IA-64 におけるローテイティング・レジスタは，スライドウィンドウ・アーキテクチャとは異なり，すべてのレジスタが常に参照できる状態になっている．またプロセッサのループ制御命令に同期した改名幅は常に 1 である．つまり，ある値を保持するレジスタ番号が増える方向に改名される場合には，レジスタ改名前に浮動小数点数レジスタ f32 に保持されている値が，レジスタ改名後に f127 の内容として参照できることになる．すなわち，ローテイティング・レジスタの名称の通り，レジスタ改名されるレジスタ群は環状に連続していると想定してよい（→第 2.3 節，図 2.1）．

同様に，プレディケートレジスタについても，すべてのレジスタが常に参照できる状態になっている．プロセッサのループ制御命令に同期した改名幅は常に 1 であり，汎用レジスタ，浮動小数点数レジスタと同時に，ループ制御命令に同期して改名が可能である．

このプレディケートレジスタは，IA-64 アーキテクチャの備えるループカウンタ（LC），エピローグカウンタ（EC）を用いることで，ソフトウェア・パイプライン処理におけるプロローグ及びエピローグのコードを生成せずに実行することができる機構を持つが，本論文においてこの機構を積極的には用いないこととする．

4.2.2 Spiral Graph と述語付き命令実行機構

レジスタ改名機構（第 2.3 節）を持つアーキテクチャに対するレジスタ割り付け方法として Spiral Graph が提案されている [HSYN98, Har00]．

Spiral Graph はスライドウィンドウ・アーキテクチャ [NNB96, INBN93] 向けに提案されたレジスタ割り付け方法で，レジスタ改名機構を傾いたグラフによって素直に表現できるという特徴を持ち，Spiral Graph 向けの Short Bridge Algorithm によって，実用的な時間内にほぼ最適な割り付けが得られる．さらにレジスタ改名幅が 1 である場合においては，多項式時間で必要レジスタ数が最小となるレジスタ割り付け結果が得られる．

しかし，条件分岐構造を含むプログラムに対する Spiral Graph を用いた，レジスタ割り付け方法については研究がすすんでいなかった．

スライドウィンドウ・アーキテクチャを備えた CP-PACS[Nak95] の単体プロセッサには，倍精度レジスタ 1 つを用いて，単精度の浮動小数点数を 2 つ格納し演算を行なえる機構が備わっており，Spiral Graph もこれに対応するための検討を行

なっていた [Har00]．プログラム中に別々に出現する単精度浮動小数点数の生存区間を組み合わせて，1つのレジスタを使う方法を提案しており，述語付き命令実行機構を備えた計算機においても，単精度浮動小数点数の生存区間を組み合わせる方法と同様の方法でレジスタを共有することが可能であろうという予想が述べられている．CP-PACS の単体プロセッサにおいては条件分岐向けの述語付き命令実行機構が備えられていないことから，その必要性も低かったため，この方法について詳細な検討や，実験による検証結果はまだ得られていなかった．

4.2.3 述語付き Spiral Graph

Spiral Graph においては，1つの実レジスタを1つの螺旋で表現していた．この実レジスタに対して，変数の生存区間を並べていくことでレジスタ割り付けを行なう．

述語付き命令実行機構を持つアーキテクチャの場合には，スケジューリング方法にもよるが，述語による命令の実行・非実行が選択できるため，一般的に条件分岐構造における then 節，else 節双方の命令を1つの実行パスに並べ，ソフトウェア・パイプライン処理を行なうことが考えられる．このとき，then 節で定義されたのち参照されるプログラム中の変数あるいは演算に必要な一時変数と，else 節で定義されたのち参照されるそれらはほとんどの場合全く異なっている．同様の形をしたプログラム，式が双方にあるとしても，変数には別の名前を与えて最適化を行なうからである．条件分岐の結果，then 節あるいは else 節いずれかにのみ存在する変数どうしで，実行時に同時に存在するものは当然同じレジスタを使うことはできない．

しかし，then 節と else 節に別々に存在する変数どうしは，同時に利用されることはない．これらの変数どうしは，コード上は同時に存在することになっているが，実際には同時に値を保持する必要はないため，同じ実レジスタを使って演算をすることが可能である．このような変数により同じ実レジスタを用いることを，レジスタを共有すると呼ぶこととする．

従来の Spiral Graph において，このような場合におけるレジスタの共有を表現することはできなかった．

本研究では，IA-64 など述語付き命令実行機構とレジスタ改名機構をあわせもつアーキテクチャに対して，Spiral Graph 上にレジスタの共有を表現する方法を導入した「述語付き Spiral Graph」（Predicated Spiral Graph）を提案した [IHYT01]．

従来の Spiral Graph における，1つの実レジスタを表現する螺旋（トラック）をそれぞれの述語の値に対応した「副トラック」（sub-track）で表現する．述語の値は，通常 1bit で表現されるから，述語1つに対して真あるいは偽の2本の副トラックとして表現される．

述語付き Spiral Graph は，図 4.1 で示されるグラフである．

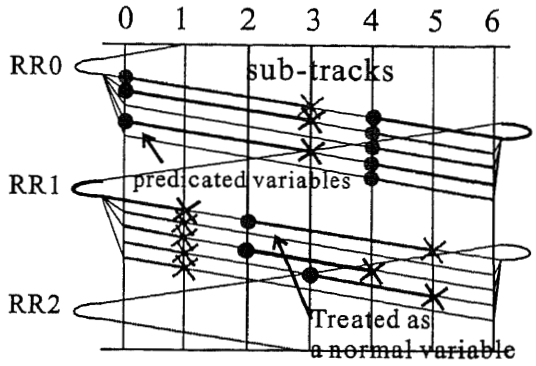

図 4.1: 述語付き Spiral Graph

条件分岐構造がネスト構造になっている場合には，述語が複数になる可能性もある．このような場合にはそれぞれの副トラックをさらに副トラックに分割することで，複数の述語による条件分岐を表現できる．

さらに，これら副トラックに割り付けられる生存区間を述語付き生存区間と呼ぶ．プログラム中の変数あるいは演算のための一時変数は，述語付き命令によって定義され参照される．プログラム実行における意味を考えると，ある変数が複数の述語付き命令によって定義されることはないから，変数の定義時の述語を変数に付加し，その述語が真の値を持つときにのみ定義，参照される生存区間を，述語付き生存区間として定義できる．

述語が常に真であるような場合，すなわち通常の生存区間とみなせる場合には，生存区間が従来の Spiral Graph において 1 つのトラックを占有したように，その生存区間が示す始点から終点までのすべての副トラックを占有する．

副トラックに生存区間が重なるように割り付けられた述語付き生存区間は，それらの和をとって通常の生存区間とみなすことにする．

定義 4.1 (始点，終点) 変数の生存区間 $v_i = [s_i, e_i)$ において，$\min s_{ij} (0 \leq s_{ij} \leq II)$ を v_i の始点，$\max e_{ij} (s_{ij} < e_{ij})$ を v_i の終点という．ただし，$v_{ij} = [s_{ij}, e_{ij})$ は v_i を構成する述語付き生存区間の集合である．

これにより，非常に効率は悪いが，通常の生存区間群に対して，従来の Short Bridge Algorithm をそのまま適用することも可能である．

4.2.4 述語付き Short Bridge Algorithm

互いにレジスタを共有可能な述語付き生存区間群から求められた通常の（述語が付かない）生存区間を用いて，Spiral Graph 上に生存区間を並べ，レジスタ割り付けをすることも可能である．

しかし，どの述語付き生存区間どうしを組み合わせて通常の生存区間とみなすのかという問題がある．さらにShort Bridge Algorithmは，生存区間をグラフ上に逐次並べ，すでに並べられた生存区間の終了ステップ番号の情報を用いて隙間を小さくしていく方法であるから，生存区間どうしを組み合わせる場合にもこれらの情報を活用する必要があると考えられる．

本研究ではこれらの問題に対して，従来のShort Bridge Algorithmを拡張し，述語付きShort Bridge Algorithm提案した．

述語付きShort Bridge Algorithmにおいては，生存区間を逐次述語付きSpiral Graphに並べていく基本的な方針に変更は無いが，ある生存区間をグラフに割り付けたとき，その生存区間と実レジスタを共有可能な生存区間を探索し，共有のための方針に基づいて副トラックを占有させる方法をとった．

アルゴリズム 4.1 述語付き *Short Bridge*

1. 変数 $v_1 1, \ldots v_N$ から始点の最も小さいものを選び，トラック *1* あるいはその副トラックに割り付ける．
2. 最後に割り付け変数と「排他」となる変数を探索し，レジスタ共有方針にもとづいて残った副トラックに割り付ける．
3. 残る変数の中から，最後に割り付けられた変数との隙間が最も小さい変数を選び，開始ステップを維持して割り付ける．
4. *2* を巻き付けていない変数がなくなるまで繰り返す．

4.2.5 レジスタ共有方針

アルゴリズム中で用いられる方針には，以下の5種類を用意した．

1. exclude allocation：
   レジスタを常に同時に割り付けない．条件分岐を考慮しない場合の割り付けと同様の結果を与える．
2. joint allocation：
   レジスタを常に同時に割り付ける．レジスタの共有率は高くなるが，螺旋全体の長さが長くなることが考えられる．
3. include allocation：
   発見されたレジスタが螺旋の長さを延ばさない時，すなわちすでに割り付けられたレジスタに完全に含まれているときに割り付ける．joint allocationの改良である．
4. look-ahead allocation：
   発見されたレジスタが螺旋の長さを延ばすかどうかは，さらに次に割り付けられる生存区間の開始位置による．include allocationの改良である．
5. slide-cover allocation：

副トラックをトラックと同様にみなし，隙間が小さくなるようにShort Bridge Algorithmを適用する方法である．制約の大きい，述語を持たない生存区間の優先順位が相対的に下がる可能性がある．

exclude allocationは，比較のために用意したレジスタ共有方針である．述語付きShort Bridge Algorithmの適用中に，割り付けられた生存区間と実レジスタを共有可能な述語付き生存区間が探索により発見されたとしても，これらで実レジスタを共有するように副トラックに割り付けることをしない．

これは述語付き生存区間の述語情報を全く用いないため，従来のSpiral Graphに対して，述語情報の無い生存区間を割り付けていく方法に等しい．アルゴリズム（述語付きShort Bridge Algorithm）上は，2で行なわれる探索自体を行なわないことに相当する．

joint allocationは，述語付きShort Bridge Algorithmの適用中に，割り付けられた生存区間と実レジスタを共有可能な述語付き生存区間が探索により発見された場合に，常に実レジスタを共有するように副トラックに割り付ける方法である．生存区間の探索順序についてはすでに割り付けられている生存区間と開始点ができるだけ近いものを優先して選択している．また，互いに干渉せずに実レジスタを共有できる述語付き生存区間が複数存在した場合にはこれらをすべて割り付けることとする．

できるかぎり実レジスタを共有するように述語付き生存区間を副トラックに割り付けていくため，レジスタの共有率が上がり，見積られる必要レジスタ数が減少するという利点がある．

しかし，共有方針に基づいて割り付けた生存区間の後尾が，もともと割り付けた生存区間の後尾より後ろになることで，次に割り付ける生存区間との間に不必要な隙間が空いてしまう，あるいは螺旋に割り付けた生存区間の全体の長さが延びてしまう可能性がある．Spiral Graphにおいて螺旋の後尾が延びてしまうことは必要レジスタ数が増大することにつながることから，問題となる．

このjoint allocationの問題点を改良したのが，include allocationである．

include allocationでは，述語付きShort Bridge Algorithmの適用中に割り付けられた生存区間と実レジスタを共有可能な述語付き生存区間が探索により発見された場合，すでに割り付けられている生存区間の終点よりも発見された述語付き生存区間の終点が前にある場合にのみ実レジスタを共有するように副トラックに割り付ける方法である．

実レジスタを共有するように生存区間を割り付けていくことでレジスタの共有率が上がり，見積もられる必要レジスタ数を減少させることができ，かつ割り付けた生存区間群の後尾が延びることを回避できる．

さらにinclude allocationにおける後尾を延ばさない条件を改良したレジスタ共有方針がlook-ahead allocationである．

look-ahead allocation では，述語付き Short Bridge Algorithm の 3 において次に割り付けられる可能性がある生存区間をあらかじめ探索しておく．述語付き Short Bridge Algorithm の 2 においてレジスタを共有して割り付ける述語付き生存区間の終点が，次に割り付けられる可能性のある生存区間の始点を越えない場合には，これが後尾を延ばさないものと判定する．

Spiral Graph と Short Bridge Algorithm によるレジスタ割り付け手法には計算量が小さいという利点があるが，look-ahead allocation においては，生存区間の探索にかかるコストにより計算量が増大するという問題点がある．

slide-cover allocation はこれまでの方針とは全く異なり，すべての副トラックを従来の Spiral Graph におけるトラックと同様に扱い，実レジスタを共有できるレジスタがなくなるまで，隙間を小さくする生存区間を順に割り付けていく方法である．

このような割り付け問題では一般的に，制約の大きい生存区間から割り付けていくほうが良い結果が得られる．slide-cover allocation においては，副トラックのすべてを占有してしまう述語が常に真の生存区間，すなわち通常の生存区間の割り付けに関する優先度が低くなる可能性がある．副トラックのすべてを占有するという制約の大きな生存区間の優先度が下がることは割り付け結果に影響を与える可能性がある．

4.3 レジスタ割り付け実験

4.3.1 実験の目的

本論文で提案した述語付き Spiral Graph と，述語付き Short Bridge Algorithm について，その有効性を検証するためにレジスタ割り付け実験を行なった．

実験では，述語を考慮しない従来の Spiral Graph をよび Short Bridge Algorithm と同等の結果を与える方法として，レジスタ共有方針の exclusive allocation を用いた．

これに対して， joint allocation，include allocation，look-ahead allocation，さらに slide-cover allocation がどれだけ有効かを示すことが実験の目的である．

4.3.2 実験方法

本実験においては，ループ中に条件分岐が 1 つ含まれるようなプログラムに対して述語付き命令実行機構を仮定した場合に想定される変数の生存区間群を乱数により生成した．IF 変換を行なったのちに，then 部および else 部の任意の命令を 1 つずつ組み合わせて同時に発行できる状態を仮定してスケジュールしたことになる．

生存区間群は，実際のプログラムでの実行状況と近い状態とするために，データ依存グラフに3種類の傾向を持たせて合計300例用意した．

1. 鎖状グラフ：
   単純な式の計算で生成されやすいグラフであり，生存区間が順序よく並んでいるのが特徴である．
2. 木状グラフ：
   通常のプロセッサの命令列から生成されやすいグラフである．式においては，1つあるいは2つの値に対して演算を行ない，1つの結果を得る命令が多いためグラフは木状になる．
3. 無傾向グラフ：
   スケジュールにより変数の利用順序はある程度決定されているものの変数の定義点および参照点が，上記のデータ依存グラフのような傾向を比較的持たないように，乱数による誤差を加えたものである．

なお，割り付け実験にはIBM PC/AT互換機（Celeron 400MHz）を用いた．

4.3.3 レジスタ数の下界との比較

一般的なレジスタ割り付け問題においては，レジスタ割り付け結果の最適解を求めるのが困難であることから，レジスタ割り付けアルゴリズムが最適な割り付けを与えているか，あるいは最適な割り付け結果にどれだけ近いかを判定するのも同様に困難である．

このため，本実験においては，必要レジスタ数の下界として考えられる値を比較のために用い，述語付きSpiral Graphと述語付きShort Bridge Algorithmによるレジスタ割り付け結果をこれら下界の値との差分で比較した．

必要レジスタ数の下界とは，必要レジスタ数としてこれよりも小さい値はありえないという指標であるから，レジスタ割り付けアルゴリズムによる割り付け結果と下界の値が一致すれば，すなわちそれが必要レジスタ数の下限であり，割り付け結果が最適であることが保証される．

本実験では

1. Case-Sensitive Lower Bound[ED95]：
   命令の実行サイクルごとに，述語の真と偽を組み合わせて必要レジスタ数の下界とする方法．
2. 山下の下界予想[Yam]
   ソフトウェア・パイプラインの実行ステージ情報をもとに述語の真と偽を組み合わせて必要レジスタ数の下界となるグラフの幅W_{max}を求める方法．このW_{max}は必要レジスタ数の下界として利用可能であると考えられており反例は見つかっていない（以降では山下の下界と書く）．

の2つの下界を比較に用いた.

4.3.4 実験結果

それぞれの割り付け方針に基づいてレジスタ割り付けを行なった結果を，レジスタ数の下界と比較した結果をグラフで示す.

グラフの横軸は，必要レジスタ数を下界に対してどれだけ多く見積ったかを表わす．グラフの縦軸は，下界との差分がその数になったサンプル数を表わす.

グラフが左側にあるほど良い結果を与えていることを示す.

鎖状のデータ依存グラフに対してレジスタ割り付けを行ない，Case-Sensitive Lower Bound と比較したのが図 4.2 である．同様に山下の下界と比較したのが図 4.3 である.

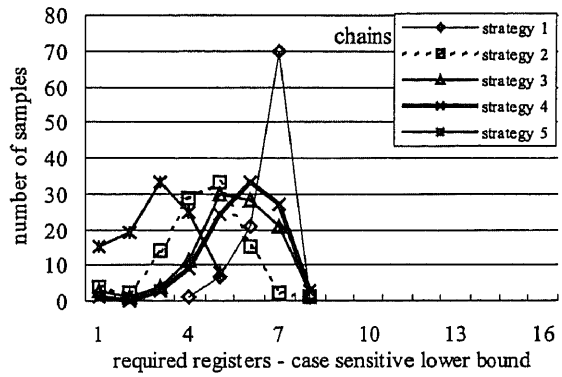

図 4.2: chains/case sensitive lower bound.

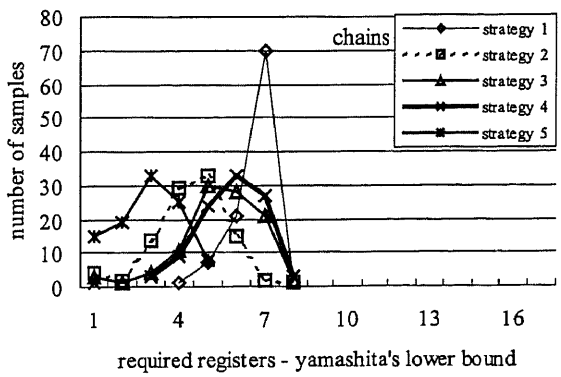

図 4.3: chains/yamashita's lower bound.

グラフ中の凡例の strategy はそれぞれレジスタ共有方針と対応している．再度，一覧に示すと次のようになる．

1. strategy 1 ： exclusive allocation
2. strategy 2 ： joint allocation
3. strategy 3 ： include allocation
4. strategy 4 ： look-ahead allocation
5. strategy 5 ： slide-cover allocation

木状のデータ依存グラフに対してレジスタ割り付けを行ない，Case-Sensitive Lower Bound と比較したのが図 4.4 である．同様に山下の下界と比較したのが図 4.5 である．

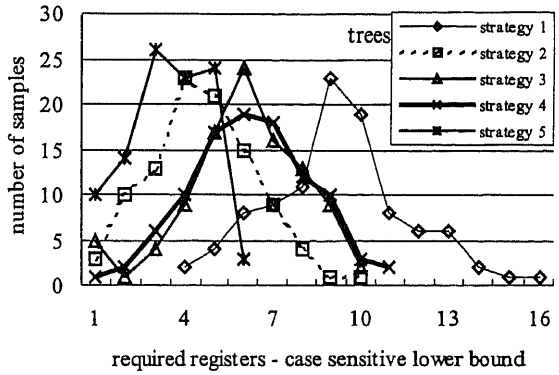

図 4.4: trees/case sensitive lower bound.

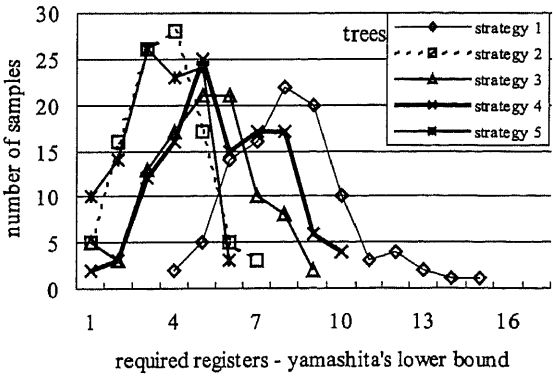

図 4.5: trees/yamashita's lower bound.

無傾向のデータ依存グラフに対してレジスタ割り付けを行ない，Case-Sensitive

Lower Bound と比較したのが図 4.6 である．同様に山下の下界と比較したのが図 4.7 である．

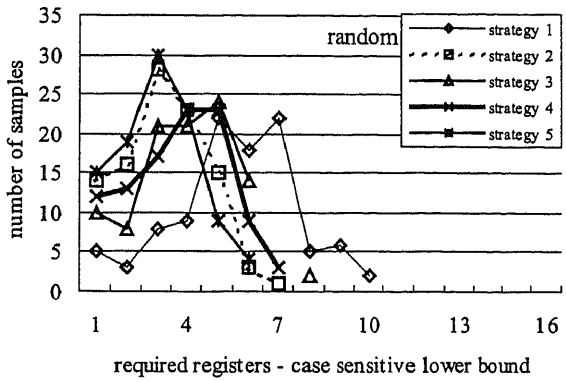

図 4.6: random/case sensitive lower bound.

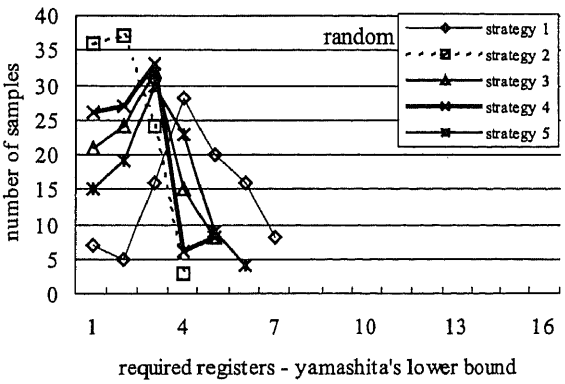

図 4.7: random/yamashita's lower bound.

4.4 レジスタ割り付け実験の考察

レジスタ割り付け実験により以下のことがわかる．

slide-cover allocation がすべてのデータ依存グラフに対して比較的良好な割り付け結果を示している．

理由として考えられるのは，プログラムにより自動生成した生存区間群が一般のプログラムと同様に，ある程度の連続性を持っているからだと考えられる．通常のプロセッサにおいてはある命令において，1 つあるいは 2 つの値から，1 つの結果

を得るものが多い．1つあるいは2つの変数が参照され生存区間が終了すると同時に，1つの変数が定義され生存区間が生成されることになる．slide-cover allocation においては，これらの連続性を保ったまま，生存区間を隙間なく実レジスタに割り付けることが可能となるからである．また，常に述語が真すなわち通常の生存区間の連続性も存在するため，述語付き Spiral Graph の副トラックの隙間が小さくなるように述語付き生存区間を優先して割り付けることになっても，割り付けた生存区間のなかに多きな隙間ができないことが考えられる．

joint allocation がこれに準じて良好な結果を与えている．この割り付け方針では，条件分岐の一方に非常に長い生存区間があった場合に問題が起こる可能性がある．今回生成した生存区間群では，条件判定の結果が真である場合と偽である場合で，実行されるコード量，生存区間の長さなどが平均化されていたことによると考えられる．

include allocation および look-ahead allocation は joint allocation を改良するためのレジスタ共有方針であるが，本レジスタ割り付け実験においては大幅な改良はみられていない．いくつかの例においては，共有条件の改良によって必要レジスタ数の見積りが，結合割り付けよりも下界に近づいているが，全体的には良い結果が得られたとは言いにくい．

なお，レジスタ割り付けに要した時間は非常に小さく，look-ahead allocation を含めてどれも 0.05CPU 秒程度（Celeron 400MHz）である．よって，複数の割り付け方法をすべて試し，最も良い結果を採用するという方法も十分に可能であろうと考えられる．

4.5 関連研究

述語付き命令実行機構が最初に搭載されたのは Cydra5 であったが，Cydra5 におけるレジスタ割り付け [DHB89, DT93] は，ソフトウェア・パイプラインを補助するために導入されたレジスタ改名機構の1つである Rotating Register File の利用に重点が置かれており，述語付き命令実行機構に特化したレジスタ割り付け方法への言及はない．

変数の生存区間を用いたレジスタ割り付け法として提案されている Cyclic Interval Graph[HGAM92a, HGAM92b] においては，条件分岐を含む場合のレジスタ割り付けについて，条件分岐構造部分をひとつのブロックとして内部でレジスタ割り付けを行ない，その結果から占有レジスタ数，ブロックに出入りするレジスタをあらかじめ決定したのち条件分岐構造の外側のレジスタ割り付けを行なう方法が示されている．これは条件分岐構造を含む場合一般について述べられた方法であり，述語付き命令実行機構や，ソフトウェア・パイプライン処理に対する割り付けに特化したものとはいえない．

干渉グラフを用いたレジスタ割り付け方法に対して，述語付き命令実行機構を

考慮したものが提案されている [ED95] が，ソフトウェア・パイプライン処理，あるいはレジスタ改名機構に対応してレジスタ割り付けを行なうのが困難なことは第 4.1 節で述べたとおりである．

第5章　結論

5.1　述語付き命令実行機構を持たないアーキテクチャにおけるレジスタ割付法

本論文において，述語付き命令実行機構を持たないアーキテクチャにおけるレジスタ割り付け方法として，**状態別干渉グラフ**を用いた方法及び**状態合併干渉グラフ**を用いた方法を提案した．

Enhanced Modulo Scheduling（EMS）[WHB92]と改良EMS[YN94]は，条件分岐を含むループを通常のアーキテクチャで効果的にモジュロ・スケジュールできるアルゴリズムだが，これに対するレジスタ割り付け方法は詳しく解析されていなかった．

EMS及び改良EMSでは，逆IF変換によって展開された複数のパイプライン・カーネル間で状態遷移しつつ実行が行なわれることから，パイプライン・カーネル内の変数の干渉を干渉グラフで表現し，パイプライン・カーネル間の状態遷移にまたがって存在する変数の同一性を有向エッジによって表現したのが状態別干渉グラフである．状態別干渉グラフは，EMS及び改良EMSの実行状態を正確に表現できるものの，EMS及び改良EMSの問題点であるコードの増大の影響を受け，近似彩色アルゴリズムによる彩色が困難になる程にグラフが巨大化することがわかった．

そこで状態別干渉グラフにおける，複数のパイプライン・カーネルの干渉グラフを重ね合わせ，グラフの大きさを縮退させた状態合併干渉グラフを提案した．状態合併干渉グラフは，干渉が過剰に考慮される場合において彩色数が増える可能性があるという不利益な条件を持つものの，グラフがEMS及び改良EMSにおけるコードの増大の影響をほとんど受けないため近似彩色アルゴリズムが適用しやすい．

本論文で提案した状態別干渉グラフと状態合併干渉グラフに対して，ノード数・彩色数・彩色時間を比較する実験を行なった．

実験には，

1. 標準的なベンチマークであるLivermore Fortran Kernel[McM86]の16番ループを，仮想アセンブリコードによる改良EMSによってスケジュールし，使われた変数の生存区間を抽出した生存区間群
2. 通常のRISCプロセッサのアセンブリコードによくみられるデータ依存グラ

フの特徴を持った変数の生存区間群

の2つの生存区間群を用いた．

この実験において状態合併干渉グラフは，グラフが正確である状態別干渉グラフとほぼ同等の彩色結果を得られることが判明した．さらに，状態合併干渉グラフは，コードの増大の影響を受けないため，効率良く彩色アルゴリズムが適用できることが判明した．

この手法の有効性が検証されたことで，EMS及び改良EMSの実用化の可能性が大きくなったといえる．また，状態別干渉グラフにおけるグラフの巨大化が，状態合併干渉グラフにおける不利益条件よりも彩色の困難さに与える影響が大きいことが判明したことで，干渉グラフの完全性よりも彩色の困難さを軽減することが有効であるという新しい知見を与えた．

5.2 述語付き命令実行機構を持つアーキテクチャにおけるレジスタ割付法

本論文において，IA-64のように述語付き命令実行機構とレジスタ改名機構を合わせ持つアーキテクチャに対するレジスタ割り付け方法として，**述語付き Spiral Graph**と，**述語付き Short Bridge Algorithm**を提案した．

述語付きSpiral Graphでは，従来のSpiral Graphにおいて1つの実レジスタを表現するためのトラックを，複数の副トラックの集合で表現した．述語付き命令によって定義，参照される変数の生存区間を述語付き生存区間として，これらの副トラックに割付ける方法をとった．

述語の値が異なる述語付き生存区間は，実レジスタを共有して使用することが可能である．実レジスタの共有のための方針を5つ提案し，これらを用いてレジスタ割付を行なう述語付きShort Bridge Algorithmを提案した．

本論文で提案した述語付きSpiral Graph，述語付きShort Bridge Algorithmに対して，必要レジスタ数を比較する実験を行なった．

実験には，鎖状，木状，無傾向の性質をもたせて生成したデータ依存グラフから抽出した生存区間群を用い，必要レジスタの下界と比較した．

この実験により，互いに異なる条件で定義・使用され，同じ実レジスタを共有して割付けることのできる変数の割付手法に指針が与えられた．

さらに，レジスタ改名機構を表現したSpiral Graphのさらなる有効性と妥当性が示された．

5.3 今後の展開

今後の研究の発展としては，本論文で提案したレジスタ割り付け手法を，条件分岐を含む場合のソフトウェア・パイプライン処理のための命令スケジューラとあわせて，実際のコンパイラ・コードスケジューラに実装することがあげらえる．標準的なベンチマークや実際に科学技術計算に用いられるプログラムによりより詳細な性能評価を行なう必要がある．

干渉グラフを用いたレジスタ割り付けにおいては，状態合併干渉グラフにおける不利益条件の正確な検出方法の確立や，近似彩色アルゴリズムの差による彩色結果の違いを詳細に検討するなどの必要がある．

Spiral Graph においては，多項式時間での最適レジスタ割り付けアルゴリズムが発見されているが，条件分岐を含む場合の述語付き Spiral Graph においても同様のアルゴリズムが存在するか精密に検討する必要がある．

また，Spiral Graph においてはレジスタ数の下界として用いられるグラフの幅 W_{max} かそれより +1 本のレジスタ数でレジスタ割り付けが行なえるという証明が得られている．条件分岐を含む場合には，比較のための下界として用いた山下の下界かそれより +1 本のレジスタ数でレジスタ割り付けが行なえるという予想がある．この証明を行なうことで，レジスタ改名機構のさらなる有効性が示せると考える．

参考文献

[AJLA95] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan.
Software pipelining.
ACM Computing Surveys, Vol. 27, No. 3, pp. 367–432, Sep 1995.

[AKPW83] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren.
Conversion of control dependence to deta dependence.
In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 177–188, 1983.

[App98] Andrew W. Appel.
modern compiler implementation in Java.
Cambridge University Press, New York, Cambridge, 1998.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon.
Improvements to graph coloring register allocation.
ACM Transactions on Programming Languages and Systems, Vol. 16, No. 3, pp. 428–455, May 1994.

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry.
Integrating register allocation and instruction scheduling for riscs.
In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 122–131, 1991.

[Ber70] C. Berge.
Graphes et hypergraphes.
Dunod, 1970.
（邦訳）
伊理正夫, 伊理由美, 岩坪秀一, 小林欣吾, 佐藤創, 星守：
グラフの理論 II,
サイエンス社, 1980.

[BSBC95] Thomas S. Brasier, Philip H. Sweany, Steven J. Beaty, and Steve Carr.

CRAIG: A Practical Framework for Combining Instruction Scheduling and Register Assignment.
In *Parallel Architectures and Compilation Techniques (PACT '95)*, 1995.

[BYA93] Gary R. Beck, David W. L. Yen, and Thomas L. Anderson.
The cydra 5 minisupercomputer: Architecture and implementation.
In *The Journal of Supercomputing*, Vol. 7 (1–2), pp. 143–180, May 1993.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein.
Register allocation via coloring.
Computer Languages, Vol. 6, pp. 47–57, 1981.

[CH90] Fred C. Chow and John L. Hennessy.
The priority-based coloring approach to register allocation.
ACM Transactions on Programming Languages and Systems, Vol. 12, No. 4, pp. 501–536, Oct 1990.

[Cha82] Gregory J. Chaitin.
Register allocation and spilling via graph coloring.
In *Proceedings of SIGPLAN '82 Symposium on Compiler Construction, ACM SIGPLAN Notices*, pp. 98–105, Jun 1982.

[CLM+95] Pohua P. Chang, Daniel M. Lavery, Scott A. Mahlke, William Y. Chen, and Wen mei W. Hwu.
The importance of prepass code scheduling for superscalar and superpipelined processors.
IEEE Transactions on Computers, Vol. 44, No. 3, pp. 353–370, 1995.

[CS99] Gang Chen and Michael D. Smith.
Reorganizing global schedules for register allocation.
In *Proceedings of 1999 ACM International Conference on Supercomputing*, pp. 408–416, Jun 1999.

[DG98] Amod K. Dani and V. Janaki Ramanan R. Govindarajan.
Register-sensitive software pipelining.
In *Proceedings of the first merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/IPDPS)*, Mar 1998.

[DHB89] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt.

Overlapped loop support in the cydra 5.
In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–38, Apr 1989.

[DKK+99] Carole Duling, Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, Jon NG, and David Sehr.
An overview of the intel ia-64 compiler.
Technology journal, Intel, Q4 1999.

[Dow93] Kevin Dowd.
High Performance Computing.
O'Reilly & Associates, 1993.
久良知 真子 訳
ハイ・パフォーマンス・コンピューティング
インターナショナル トムソン・パブリッシング ジャパン
1994.

[DT93] James C. Dehnert and Ross A. Towle.
Compiling for the cydra 5.
In *The Journal of Supercomputing*, Vol. 7 (1–2), pp. 181–228, May 1993.

[ED95] Alexandre E. Eichenberger and Edward S. Davidson.
Register allocation for predicated code.
In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO-28)*, pp. 180–191, 1995.

[EDA] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham.
Minimum register requirements for a modulo schedule.
In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27).*

[ELM95] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol.
The meeting graph : a new model for loop cyclic register allocation.
In *Proceedings of the 5th Workshop on Compilers for Parallel Computers (CPC95)*, pp. 503–516, Jun 1995.

[GA96] Lal George and Andrew W. Appel.
Iterated register coalescing.
ACM Transactions on Programming Languages and Systems, Vol. 18, No. 3, pp. 300–324, May 1996.

[Har00] 秋川友宏.
スライドウィンドウを考慮したレジスタ割付の研究.
博士課程 工学研究科 博士論文, 筑波大学, 2000.

[Hew90] Hewlett Packard.
PA-RISC 1.1 Architecture and Instruction Set Reference Manual, 1990.

[HG83] John L. Hennessy and Thomas Gross.
Postpass code optimization of pipeline constraints.
Programming Languages and Systems, Vol. 5, No. 3, pp. 422–448, 1983.

[HGAM92a] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji.
A register allocation framework based on hierarchical cyclic interval graphs.
In *Lecture Notes in Computer Science (LNCS)*, Vol. 641, pp. 176–191. Springer-Verlag, 1992.

[HGAM92b] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji.
Register allocation using cycric interval graphs: A new approach to an old problem.
Technical report, Advanced Computer Architecture and Program Structures Group Technical Memo 33, McGill University, 1992.

[HSYN98] 秋川友宏, 添野元秀, 山下義行, 中田育男.
スライドウィンドウを考慮したレジスタ割付.
情報処理学会論文誌, Vol. 39, No. 9, pp. 2684–2694, 1998.

[Hu] Ping Hu.
The techniques for software pipelining loops with conditional constructs: A survey.
http://www-rocq.inria.fr/ hu/.

[HYN99] 秋川友宏, 山下義行, 中田育男.
スライドレジスタ割付問題の厳密解法.
情報処理学会論文誌, Vol. 40, No. 9, pp. 3524–3536, 1999.

[IHYN98] 糸賀裕弥, 秋川友宏, 山下義行, 中田育男.
条件分岐を含む場合の最適なソフトウェア・パイプライニング.
情報処理学会第56回（平成10年前期）全国大会 講演論文集（1）, pp. 44–45, 1998.

[IHYN99] 糸賀裕弥, 秋川友宏, 山下義行, 中田育男.
条件分岐を考慮したソフトウェア・パイプライニングにおけるレジスタ割付.
並列処理シンポジウム JSPP'99, pp. 39–46, 1999.

[IHYN02] 糸賀裕弥, 秋川友宏, 山下義行, 中田育男.
条件分岐を考慮したソフトウェアパイプラインにおけるレジスタ割付け.
電子情報通信学会論文誌 DI, Vol. 85-D-I, No. 1, 2002.
（採録決定済）.

[IHYT01] Hiroya Itoga, Tomohiro Haraikawa, Yoshiyuki Yamashita, and Jiro Tanaka.
Register allocation for software pipelining with predication using spiral graph.
In *Proceedings of the International Symposium on Future Software Technology (ISFST2001)*, pp. 58–65, 2001.

[Ike00] 池井満.
IA-64プロセッサ基本講座.
オーム社, 2000.

[INBN93] 位守弘允, 中村宏, 朴泰佑, 中澤喜三郎.
スライドウィンドウ方式による擬似ベクトルプロセッサ.
情報処理学会論文誌, Vol. 34, No. 12, pp. 2612–2623, 1993.

[Int99] Intel.
IA-64 Application Developer's Architecture Guide, May 1999.

[Int00a] Intel.
Intel Itanium Architecture Software Developer's Manual Vol. 1 rev. 1.1: Application Architecture, Jul 2000.

[Int00b] Intel.
Intel Itanium Architecture Software Developer's Manual Vol. 2 rev. 1.1: System Architecture, Jul 2000.

[Int00c] Intel.
Intel Itanium Architecture Software Developer's Manual Vol. 3 rev. 1.1: Instruction Set Reference, Jul 2000.

[Int00d] Intel.
IntelR Itanium? Architecture Software Developer's Manual Vol. 4 rev. 1.1: Itanium processor Programmer's Guide, Jul 2000.

[Int01] Intel.

Intel Itanium Processor Reference Manual for Software Optimization, Nov 2001.

[IY99] 糸賀裕弥, 山下義行.
複数パスのソフトウェア・パイプラインにおけるレジスタ割付.
情報処理学会第62回（平成13年後期）全国大会 講演論文集（1）, pp. 153–154, 1999.

[IYN99] 糸賀裕弥, 山下義行, 中田育男.
条件分岐向けソフトウェア・パイプラインスケジューラの実装.
情報処理学会第59回（平成11年後期）全国大会 講演論文集（1）, pp. 223–224, 1999.

[JA91] Reese B. Jones and Vicki H. Allan.
Software pipelining: An evaluation of enhanced pipelining.
In *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO-24)*, pp. 82–92, 1991.

[Jin99] 神保進一.
最新マイクロプロセッサテクノロジ（増補改訂版）.
日経BP社, 1999.

[Lam88] Monica Lam.
Software pipelining : an effective scheduling technique for vliw machines.
In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 318–328, 1988.

[Man85] Bennet Manvel.
Extremely greedy coloring algorithms.
In *Graphs and Applications, Proceedings of the First Colorado Symposium on Graph Theory*, 1985.

[McM86] Frank McMahon.
The livermore fortran kernels: A computer test of the numerical performance range.
Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, 1986.

[MJ01] Dragan Milicev and Zoran Jovanovic.
A finite state machine based formal model of software pipelined loops with conditions.
International Journal of Computer Research, Vol. 10, No. 1, pp. 11–20, 2001.

[Nak95] 中澤喜三郎.
計算機アーキテクチャと構成方式.
朝倉書店, 1995.

[Nak99] 中田育男.
コンパイラの構成と最適化.
朝倉書店, 1999.

[NNB96] 中澤喜三郎, 中村宏, 朴泰佑.
超並列計算機 CP-PACS のアーキテクチャ.
情報処理, Vol. 37, No. 1, pp. 18–28, 1996.

[NYO96] 中田育男, 山下義行, 小柳義男.
超並列計算機 CP-PACS のソフトウェア.
情報処理, Vol. 37, No. 1, pp. 29–37, 1996.

[Pin93] Shlomit S. Pinter.
Register allocation with instruction scheduling: A new approach.
In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 248–257, 1993.

[PS91] Joseph C. H. Park and Mike Schlansker.
On predicated execution.
Technical report, Technical Report HPL-91-58, Hewlett Packard Laboratories, 1991.

[Rau94] B. Ramakrishna Rau.
Iterative modulo scheduling: An algorithm for software pipelining loops.
In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, pp. 63–74, Nov 1994.

[RF93] B. Ramakrishna Rau and Joseph A. Fisher.
Instruction-level parallel processing: History, overview, and perspective.
In *The Journal of Supercomputing*, Vol. 7 (1–2), pp. 9–50, May 1993.

[RLTS92] B. Ramakrishna Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker.
Register allocation for software pipelined loops.
In *Proceedings of the SIGPLAN Conference on Programing Language Design and Implementation (PLDI)*, pp. 283–299, Jun 1992.

[Sam95] 寒川光.
RISC 超高速化プログラミング技法.
共立出版, 1995.

[Sas89] 佐々政孝.
プログラミング言語処理系.
岩波講座ソフトウェア科学. 岩波書店, 1989.

[SL96] Mark G. Stoodley and Corinna G. Lee.
Software pipelining loops with conditional branches.
In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29)*, pp. 262–273, Dec 1996.

[Top] TOP500 list for novermber 1996.
http://www.top500.org/lists/1996/11/.

[WHB92] Nancy J. Warter, Grant E. Haab, and John W. Bockhaus.
Enhanced modulo scheduling for loops with conditional branches.
In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)*, pp. 170–192, 1992.

[WKEE94] Jian Wang, Andreas Krall, M. Anton Ertl, and Christine Eisenbeis.
Software pipelining with register allocation and spilling.
In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, pp. 95–99, 1994.

[WLmWH93] Nancy J. Warter, Daniel M. Lavery, and Wen mei W. Hwu.
The benefit of predicated execution for software pipelining.
In *Proceedings of the 26th Hawaii International Conference on System Sciences (HICSS-26)*, Vol. 1, pp. 497–506, Jan 1993.

[WMmWHR93] Nancy J. Warter, Scott A. Mahlke, Wen mei W. Hwu, and B. Ramakrishna Rau.
Reverse if-conversion.
In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 290–299, 1993.

[WPP95] Nancy J. Warter-Perez and Noubar Partamian.
Modulo scheduling with multiple initiation intervals.
In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO-28)*, pp. 111–118, Nov 1995.

[Yam] 山下義行.

述語実行と rotating register における W_{max} についての考察.
(in private communication.).

[YN94] 山下義行, 中田育男.
ループ中に条件分岐を含む場合の最適なソフトウェア・パイプライニング.
並列処理シンポジウム JSPP'94, pp. 17–24, 1994.

[ZC91] Hans Zima and Barbara Chapman.
Supercompilers for Parallel and Vector Computers.
Addison-Wesley, 1991.
村岡 洋一 訳
スーパーコンパイラ
オーム社
1995.

[日立] 日立製作所.
Short bridge algorithm 割付器の実装と評価.
HLS971107-01.

謝辞

本研究を進めるうえで多くの御指導をいただいた佐賀大学理工学部知能情報システム学科山下義行教授，法政大学情報科学部コンピュータ科学科中田育男教授，静岡大学情報学部情報科学科秡川友宏助手に深い感謝をいたします．

筑波大学電子・情報工学系田中二郎教授には親身に研究の御指導，博士論文の御指導をいただきました．深く深く感謝いたします．筑波大学機能工学系白川友紀教授には博士論文の御指導をいただき深く感謝いたします．

図書館情報大学図書館情報学科中井央講師，筑波大学電子・情報工学系志築文太郎助手，同三浦元喜助手には，研究にあたりまして多くの助言をいただきました．

筑波大学旧プログラミング言語研究室，旧言語処理系研究室の田井秀樹さん，品野竜太さん，中家鉄雄さん，筑波大学インタラクティブプログラミング研究室の亀山裕亮さん，酒寄保隆さん，Simona Mirela Vasilache さん，飯塚和久さん，小川徹さん，劉学軍さんには研究にご助言をいただき，研究室にて様々な助力をいただきました．

図書館情報大学図書館情報学科の宇陀則彦助教授，長谷川秀彦助教授，坂口哲男助教授，歳森敦助手，及び図書館情報大学総合情報処理センター主任専門職員鈴木廣文氏には図書館情報大学事務職員として赴任しました際に研究への多大なる助力をいただきました．

茨城県工業技術センターの村上和雄センター長，本多敏士副センター長，高島茂雄副センター長，舘義雄技術交流室長，石川友彦主任研究員には茨城県工業技術センター流動研究員として赴任しました際に研究への多大なる助力をいただきました．

成田高等学校の星野光徳教諭，金杉正明教諭には様々なご助言をいただきました．

成田高等学校より親しくさせていただいている勝股正義さん，桑田裕久さん，小林（杉森）さやかさん，鈴木祥子さん，豊島祐樹さん，中西（黒田）曜子さん，二瓶ゆりかさん，福田泰子さん，宮永厚さん，村田美由貴さん，筑波大学より親しくさせていただいている岡田潤さん，恩田朋哉さん，市山洋乃さん，南雲淳さん，福士（境）由子さん，藤原清司さん，光延秀樹さん，茨城県工業技術センターより親しくさせていただいている服部真智子さんには，様々なご助言をいただき，精神的に大きな助けとなっていただきました．

そして，わたしを育ててくれた両親と祖母，妹の範衣にあらためて感謝し，博士論文を亡祖父 糸賀二郎に捧げます．

発表論文リスト

1）論文（査読あり）

糸賀裕弥，秋川友宏，山下義行，中田育男：
「条件分岐を考慮したソフトウェア・パイプラインにおけるレジスタ割付」，
電子情報通信学会論文誌 D-I, Vol. 85-D-I, No. 1, 2002（採録決定済み）

2）国際会議（査読あり）

Hiroya ITOGA, Tomohiro HARAIKAWA, Yoshiyuki YAMASHITA, Jiro TANAKA:
"Register Allocation for Software Pipelining with Predication using Spiral Graph,"
Proceedings of the International Symposium on Future Software Technology (ISFST2001), pp. 58-65, Zheng Zhou, China, Nov. 5-8, 2001

3）国内会議（査読あり）

糸賀裕弥，秋川友宏，山下義行，中田育男：
「条件分岐を考慮したソフトウェア・パイプライニングにおけるレジスタ割付」，
並列処理シンポジウム JSPP '99，論文集 pp. 39—46，1999 年.

4）学会発表

糸賀裕弥，山下義行，中田育男：
「条件分岐を考慮したループ並列化の 1 手法」，
情報処理学会　第 54 回（平成 9 年前期）全国大会，講演論文集（1） pp. 351—352，
1997 年.

糸賀裕弥，秋川友宏，山下義行，中田育男：
「条件分岐を含むループの最適なソフトウェア・パイプライニング」，
情報処理学会　第 56 回（平成 10 年前期）全国大会，講演論文集（1） pp. 44—45，
1998 年.

糸賀裕弥，山下義行，中田育男：
「条件分岐向けソフトウェア・パイプラインスケジューラの実装」，
情報処理学会　第 59 回（平成 11 年後期）全国大会，講演論文集（1） pp. 223—224，
1999 年.

糸賀裕弥，山下義行：
「複数パスのソフトウェア・パイプラインにおけるレジスタ割付」，

情報処理学会　第62回（平成13年前期）全国大会，講演論文集（1） pp. 153—154，2001年.

5）研究会発表

糸賀裕弥，秡川友宏，山下義行，中田育男：
「条件分岐を含むループの最適なソフトウェア・パイプライニング」，
科研費特定領域研究A「ソフトウェア発展」A04班研究会，1998年5月.

糸賀裕弥，山下義行，中田育男：
「条件分岐向けソフトウェア・パイプラインスケジューラの1方法」，
科研費特定領域研究A「ソフトウェア発展」A04班研究会，1999年11月.

6）修士論文

糸賀裕弥，山下義行，中田育男：
「条件分岐を考慮したループ並列化の1手法」
筑波大学博士課程工学研究科　修士論文，平成9年.

以上.（2002年1月8日現在）