

命令レベル並列プロセッサにおける
レジスタ割付けに関する研究

Studies on Register Allocation for
Instruction-Level Parallel Processors

2008 年 2 月

早稲田大学大学院 理工学研究科

情報・ネットワーク専攻 ソフトウェア工学研究

片岡 正樹

目次

第1章	はじめに	1
第2章	本論文が対象とする並列計算機モデル	7
2.1	命令依存関係	7
2.2	並列計算機の実行モデル	10
第3章	本論文が用いるデータ表現	13
3.1	プログラムの情報	13
3.1.1	命令間の依存関係	13
3.1.2	仮想レジスタ間の干渉度	14
3.2	プログラム情報の表現	15
3.2.1	SSA形式	15
3.2.2	GPDG	16
3.2.3	GPDGの例	17
3.2.4	エッジコスト付干渉グラフ	19
3.2.5	エッジコスト付干渉グラフの例	19
第4章	プロセッサ並列度を考慮したレジスタ割付け技法	23
4.1	まえがき	23
4.2	本手法の概要	24
4.2.1	最大干渉数低減操作	25
4.2.2	プレスケジューリングおよびコード生成	27
4.3	本手法の流れ	29
4.3.1	GPDGの作成	31
4.3.2	レジスタ生存グラフの作成	41
4.3.3	最大干渉数低減操作	44

4.3.4	プレスケジューリング	50
4.4	実験	55
4.5	関連研究	59
4.5.1	並列化干渉グラフを用いたレジスタ彩色法	59
4.5.2	Integrated Code-Scheduling	64
4.5.3	レジスタ生存グラフを用いた手法	65
4.6	あとがき	66
第5章	コンパイル時間と性能の両立を図ったレジスタ割付け技法	67
5.1	まえがき	67
5.2	本手法の概要	68
5.2.1	干渉グラフの改善	69
5.2.2	彩色方法の改善	69
5.2.3	スピル方法の改善	70
5.3	本手法の流れ	71
5.3.1	生存区間解析	72
5.3.2	エッジコスト付干渉グラフの作成	77
5.3.3	過彩色	80
5.3.4	スピル付併合	83
5.4	実験	91
5.4.1	コンパイル速度の比較	92
5.4.2	生成コードの性能比較	94
5.5	関連研究	96
5.5.1	グラフ彩色法	96
5.5.2	Optimistic Coalescing	97
5.5.3	リニアスキャン	98
5.5.4	リザーブドレジスタ	99
5.6	あとがき	100
第6章	おわりに	103
	謝辞	105
	研究業績	111

目 次

2.1	5つの依存を持つサンプルコード	9
2.2	実行制御の表現と実行モデル	12
3.1	SSA 変換の例	16
3.2	GPDG の例	18
3.3	エッジコスト付干渉グラフの例	20
4.1	レジスタ生存グラフの例	26
4.2	機能ユニット数の違いによる物理レジスタ不足の例	28
4.3	本手法の流れ	30
4.4	サンプルコード	37
4.5	GPDG の生成	40
4.6	レジスタ生存グラフの生成	43
4.7	干渉数低減操作 1 の例	46
4.8	干渉数低減操作 2 の例	47
4.9	最大干渉数低減操作を施したレジスタ生存グラフ	49
4.10	プレスケジューリングを施したレジスタ生存グラフ	52
4.11	プレスケジューリングまでの変更を加えた GPDG	53
4.12	レジスタ数 8 の場合の実行サイクル数比率	56
4.13	レジスタ数 32 の場合の実行サイクル数比率	57
4.14	レジスタ数 32 の場合の使用レジスタ数比率	58
4.15	並列化干渉グラフがほぼ完全グラフになるサンプルコード	61
4.16	干渉グラフの例	62
4.17	並列化干渉グラフの例	63
4.18	Integrated Code-Scheduling の流れ	64
5.1	本手法の流れ	71

5.2	生存区間解析が苦手とする構造	74
5.3	サンプルコード	75
5.4	生存区間解析結果	76
5.5	エッジコスト付干渉グラフの実装例	79
5.6	エッジコスト付干渉グラフの例	80
5.7	過彩色の例	83
5.8	同色ノードの併合の実装例	86
5.9	同色ノードの併合例	87
5.10	異色ノード併合の例	90
5.11	コンパイルにかかった平均時間比率	93
5.12	生成コード 1000 回実行の時間比率	95
5.13	リザーブレジスタ方式での彩色例	100

表 目 次

4.1	レジスタ数 8 の場合の実行サイクル数	56
4.2	レジスタ数 32 の場合の実行サイクル数	57
4.3	レジスタ数 32 の場合の使用レジスタ数	58
5.1	生存区間解析の経過	75
5.2	コンパイルにかかった平均時間 (ミリ秒)	92
5.3	生成コード 1000 回実行の時間 (ミリ秒)	94

第1章 はじめに

アプリケーションの大規模化に伴い、プログラムを高速に実行することが求められるようになってきている。与えられたプログラムを高速に実行するためには、大きく分けて3つのアプローチが存在する。1つ目はハードウェアの高クロック化、2つ目はハードウェアの並列化、最後はコンパイラによる最適化である。これまでは、1つ1つの処理を細かくして多段処理にしたり、ハードウェアを小型化したりすることによって高クロック化を実現してきた。しかし、最近ではそれらとは反比例の関係にあるリーク電流などによる発熱量といった問題から、コストがかからない方法でハードウェアを冷却することは困難になってきている。特に、CPUに関しての高クロック化は、ほぼ頭打ちの状態であり、パフォーマンスを維持したまま省電力化などができるようにならない限り、高クロック化によるプログラムの実行の高速化を図ることは難しいといえる。そのため、今以上にプログラムを高速に実行するためには、ハードウェアを並列化するか、もしくはコンパイラによって今まで以上に最適化をかけなければならない。ただし、ハードウェアの並列化によって高速化を図るためには、並列化されたハードウェアの性能を最大限に発揮できるように、コンパイラによってプログラムを切り分ける必要があるため、結局はプログラムを高速実行するためには、コンパイラによる最適化がもっとも重要だといえる。

コンパイラによる最適化の中でも、メモリ最適化の1つであるレジスタ割付けと、プログラムの並列性の抽出を行うコードスケジューリングは特に重要である。レジスタ割付けが重要なのは、データへの高速なアクセスが可能なレジスタは有限個だからである。一方、コードスケジューリングが重要なのは、ハードウェアが一度にフェッチすることができる命令の範囲には限りがあるからである。命令の順番を並べ替えることによって、この

一度にフェッチすることができる命令の範囲の中に、より多くの同時実行可能な命令が含まれるようにして、ハードウェアのサポートを行っているからである。

レジスタ割付けとコードスケジューリングは、どちらも欠かすことのできない重要な最適化であるにもかかわらず、お互いに相性が悪いという問題点がある。なぜなら、コードスケジューリングはプログラムの並列性を上げようとしているのに対して、レジスタ割付けは一般的にプログラムの並列性を下げてしまう傾向があるからである。これは、レジスタの数は有限であり、すべてのデータをレジスタに割付けられない場合には、データをレジスタ以外のメモリに退避するため、そのデータを用いる演算の実行が遅くなるからである。言い換えれば、レジスタの数が無限であるならば不要である仮想的な命令間の依存が、レジスタが有限であるために必要になるということである。レジスタ割付けをコードスケジューリングより先に行った場合には、プログラムの並列性を高めることができなくなってしまふことが問題となる。逆に、コードスケジューリングをレジスタ割付けより先に行った場合には、コードスケジューリングがせっかく抽出したプログラムの並列性を、この仮想的な依存を挿入することによる、メモリとのやり取りが発生によって、予想以上に並列度を低くしてしまうことが問題となる。つまり、どちらの最適化を先に行ったとしても、レジスタ割付けの動作がコードスケジューリングの動作を阻害してしまう関係があるということである。

このことから、プログラムの並列性をできるだけ落とさないレジスタ割付けの研究は重要である。プログラムの並列性をできるだけ落とさないようにするためには、レジスタ割付けとコードスケジューリングを同時に行うアプローチと、協調動作させるアプローチが存在する。ただ、前者のアプローチは、命令の最も良い移動箇所を求める問題も、データの最も良いレジスタへ割付け方を求める問題も NP 完全問題であるため、同時解法は非常に困難であり、実用的な時間で最適解に近い解を求められるヒューリスティクスは、現時点では提案されていない。それに対して、後者の協調動作というアプローチに関しては、さまざまなヒューリスティクスが提案されている。その中でも、プログラムの依存関係を

表すことができるグラフ構造を用いた手法は、プログラムの高い並列性を保持したままレジスタ割付けが行えることで知られている。

しかし、このプログラムの依存関係を表すことができるグラフ構造を用いた手法には、プログラムの構造によっては、レジスタ割付け完了時にレジスタが足りているように見えても、実行コードを生成する段階においてレジスタが不足する可能性があるという問題点があった。この問題が発生する原因は、機能ユニットの個数が無限にあると仮定しているレジスタ割付けの完了時には、同時に実行されると判断していた命令が、実行コードを生成する段階において、対象アーキテクチャの機能ユニットの個数が不足することにより同時に実行できず、コードの発行に失敗、もしくはレジスタのデータを破壊する場合があるからである。この問題を解決するために、本手法ではレジスタ割付けの段階で、対象アーキテクチャの機能ユニット数を保持しておき、仮想的な機能ユニットに対して命令の割付けを行うことで、事前にレジスタが不足するプログラムの構造を検出し、排除している。これによって、レジスタが不足する構造を持ったプログラムにおいて、平均で約8%の性能向上を確認している。

また一方で、レジスタ数の少ないアーキテクチャでは、プログラム中で使用されるデータの多くをレジスタに割付けることができず、他のアーキテクチャに比べてメモリへのアクセス回数が多い傾向がある。つまり、高速実行にはあまり適さないアーキテクチャではあるが、x86 アーキテクチャのようなレジスタ数の少ないアーキテクチャは、いまだに世の中で広く使われている。よって、レジスタ数の少ないアーキテクチャでも高速実行が可能なように、レジスタ割付けを行うことはいまだに重要である。なぜなら、生成される実行コードの対象がレジスタ数の少ないアーキテクチャである場合は、一般的に生成された実行コードの性能が高ければコンパイルの速度が非常に遅く、コンパイラの速度が速ければ生成された実行コードの性能が低いという問題点があるからである。前者の問題は、レジスタ割付けが失敗するたびにメモリとのやり取りを行うコードの挿入が発生し、プログラムの構造が変化するため、レジスタ割付けで用いるグラフ構造をプログラムの解析から

4 第1章 はじめに

やり直して再作成しなおすことが原因である。また、後者の問題は、速度を重視するために性能を除外視し、最適解から遠い局所解に陥りやすくなっていることが原因である。

このことから、レジスタ数の少ないアーキテクチャ向けであっても、コンパイル速度が速く、生成される実行コードの性能も高いレジスタ割付けの研究は重要である。この2つを同時に実現するためには、プログラム中のデータをレジスタに割付けることができず、メモリとのやり取りを行う必要があっても、レジスタ割付けで用いるグラフ構造の再作成を行わずにすむ手法が必要である。

以上のことを行うためには、従来のグラフ構造が保持する情報、つまり、あるデータがレジスタを用いる区間は、他のデータがレジスタを用いる区間と重なりがあるかないかという1ビットの情報だけでは不十分である。そこで、この1ビットだけの情報の代わりに、同じレジスタを交代で用いた場合にどれほどメモリとのやり取りを行う必要があるかという整数値を保持するようにし、また各データのレジスタを用いる区間を保持するように、グラフ構造を拡張する必要がある。このようにすることで、あるデータがレジスタに割付けられない場合には、1回のグラフ構造の変形と少々の演算を行うのみで、プログラムにメモリとのやり取りを行うコードを挿入することが可能となり、従来手法より高速なレジスタ割付けが実現できる。また、メモリとのやり取りが必要な部分を解析的に求めることも可能であり、高速なレジスタ割付けが可能な従来手法より、高い性能の実行コードを生成することも可能になる。これによって、x86用のC言語コンパイラの1つであるcygwin gcc-3.4.4に本手法を組み込んだ場合、組み込まなかったgcc-3.4.4より高速にレジスタ割付けが可能で、かつ、生成された実行コードの性能でも最高で約6%の性能向上を確認している。

上記に基づいて、本論文の構成は以下のようになっている。第1章において、研究の背景および目的を述べる。第2章において、本論文が対象とする計算機のモデルについて述べる。第3章では、本論文で用いている定義やグラフ構造について述べる。第4章と第5章は本論文の中核である、上記のアプローチについて詳細を述べる。特に、第4章ではブ

プログラムの並列性をできるだけ落とさないレジスタ割付けの研究について，第5章ではレジスタ数の少ないアーキテクチャ向けのレジスタ割付けの研究について，手法の特徴や適用例を述べる．また，想定した環境下において実験を行い，その結果に対する考察も述べる．最後に第6章で本論文をまとめる．

第2章 本論文が対象とする並列計算機モデル

2.1 命令依存関係

一般的に、プログラムの構成要素である命令間には依存関係が存在する。2つの命令間に依存関係がある場合、その後続命令は先行命令より先に実行することはできないが、依存関係がない場合は、どのような順番で実行してもかまわない。このように依存関係とは命令の実行順序の制約を意味し、依存関係がない命令間には並列性が存在する。

分岐によって発生する制御依存の他に、コンパイラが静的に解析できるデータ依存には、その性質上から以下の5種類に分けられる。

- 真依存 (true dependency) : 命令 a がデータ X を定義し、命令 b がそのデータ X を使用する場合に、命令 b は命令 a に対して真依存関係を持つ。例えば、図 2.1 の S_2 の右辺である `load A[i]` は、 S_1 の左辺である `store A[i]` と真依存関係を持っている。
- 逆依存 (anti-dependency) : 命令 a がデータ X を使用し、命令 b がそのデータ X を再定義する、つまり上書きする場合に、命令 b は命令 a に対して逆依存関係を持つ。例えば、図 2.1 の S_3 の左辺である `store A[i]` は、 S_2 の右辺である `load A[i]` と逆依存関係を持っている。
- 出力依存 (output dependency) : 命令 a がデータ X を定義し、命令 b がそのデータ X を再定義する、つまり上書きする場合に、命令 b は命令 a に対して出力依存関係を持つ。例えば、図 2.1 の S_3 の左辺である `store A[i]` は、 S_1 の左辺である `store A[i]`

と出力依存関係を持っている。

- 入力依存 (input dependency) : 命令 a がデータ X を使用し, 命令 b がそのデータ X を使用する場合に, 命令 b は命令 a に対して入力依存関係を持つ。例えば, 図 2.1 の S_5 の右辺である load B[i] は, S_1 の右辺である load B[i] と入力依存関係を持っている。
- 制御依存 (control dependency) : 命令 a が分岐命令であり, 命令 b が命令 a の分岐先の命令である場合に, 命令 b は命令 a に対して制御依存関係を持つ。例えば, 図 2.1 の S_1 から S_5 の命令は, ループの条件文と制御依存関係を持っている。

この中で, 入力依存のみ命令の実行順序の制約に影響を与えない。逆依存と出力依存については, 再定義することによって発生する依存であるため, 変数のリネーミングといったプログラム変換によって, 除去可能である場合が多い。また, 制御依存については, 精度の高い分岐予測を行うことや, 分岐先にある命令を両方とも実行して分岐が成立した命令だけ実行結果を残すという投機的実行などにより, 緩和が可能である。

```

                for (i=0; i<IMAX; i++){
S1            A[i] = B[i];
S2            C[i] = A[i];
S3            A[i] = C[i-1];
S4            B[i-1] = D[A[i]];
S5            D[i] = B[i];
                }

```

図 2.1: 5 つの依存を持つサンプルコード

また、プログラムにループが存在する場合、イタレーション内に存在する命令は、ループの終了条件を満たすまで繰り返し実行される。イタレーションを構成する命令間に存在するデータ依存は、以下の 2 種類に分けられる。

- ループ独立依存 (loop-independent dependency) : 同じイタレーションのみに存在する依存関係。
- ループ運搬依存 (loop-carried dependency) : 異なるイタレーション間に存在する依存関係。

これらの依存関係は、前述の 4 つの依存関係とは独立であるため、ループ内では 8 種類の依存関係に分類できる。ループ運搬依存の異なるイタレーション間での依存関係とは、例えばループ変数 $i=k$ のときに定義されたデータ X を、 $i=k+1$ のときに使用するような関係のことをいう。

ここでは、ループ運搬依存が1つもないループのことを DOALL 型ループ、ループ運搬依存があるループのことを DOACROSS 型ループと呼ぶことにする。

2.2 並列計算機の実行モデル

プログラム実行のモデルには、代表的なものとして以下の6つが挙げられる。

1. SISD (Single Instruction / Single Data)

これは1つの命令で1つのデータを扱う処理方式である。処理システムは1つであり、半順序関係であるプログラムの命令列を全順序関係に変換し逐次実行するイン・オーダ実行と、半順序関係のまま実行可能なものから実行するアウト・オブ・オーダ実行がある。この SISD では、実行順序制御として次のような条件を持つ。

(a) プログラムカウンタ (PC) は1つのみである。

(b) PC が指し示す命令は1つのみである。

2. スーパースカラ (superscalar)

SISD では1つであった処理システムを複数にすることで、依存関係のない命令を同時に実行する。つまり、未実行の命令を溜めておく機構があり、その中から実行可能な命令を処理システムの数だけ選択し、アウト・オブ・オーダ実行を行う。

3. VLIW (Very Long Instruction Word)

SISD の条件 (b) が無い処理方式である。つまり、PC は1つのままだが指し示す命令が複数になっても良い。名前のとおり1つの命令語が長く、1つの命令語は複数個の命令から構成されている。プログラムは、この命令語列に全順序関係を持たせることで構成される。また1つの命令語に含まれる複数個の命令は同時に実行されるため、その命令間には依存関係があってはならない。

4. SIMD (Single Instruction / Multiple Data)

1つの命令で複数のデータを扱う処理方式である。お互いに依存関係のない大量のデータに対して、同じような操作を行う場合に適している。

5. MIMD (Multiple Instruction / Multiple Data)

SISD の条件 (a) が無い処理方式である。つまり、PC が複数個存在しても良い。プログラムは、元のプログラムの部分集合であるタスク内の命令列に全順序関係、各タスク間に半順序関係を持たせることで構成される。各 PC は、それぞれに割り振られたタスク内にある 1 つの命令を指し示す。

6. データフロー (dataflow)

SISD の条件 (a),(b) の両方が無い処理方式である。つまり、プログラムを半順序関係のまま実行する方式である。PC では、半順序関係にあるプログラムの実行制御ができないため、最も単純なデータフローモデルでは、各命令に次命令のポインタを持たせている。

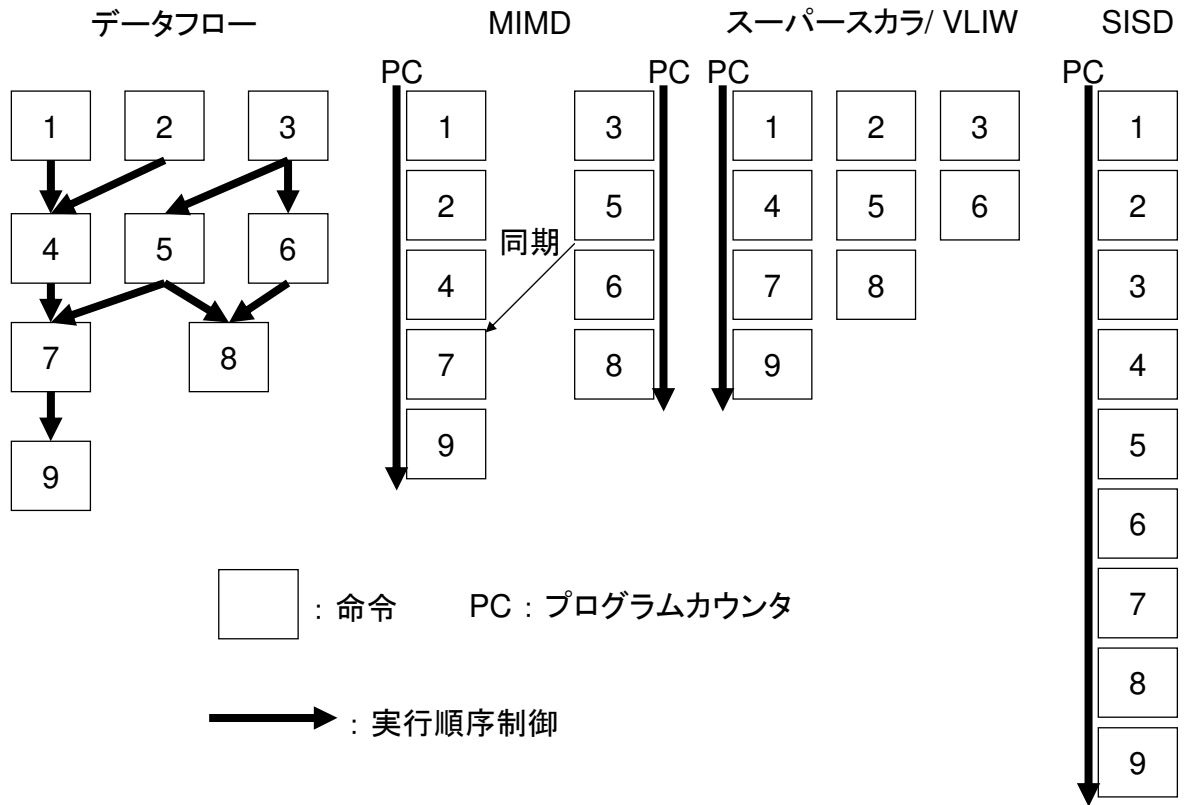


図 2.2: 実行制御の表現と実行モデル

図 2.2 にそれぞれの様子を示す．一般に並列処理と言う場合は，MIMD から左側の実行モデルにおける処理を指す．これは必ずしも命令レベル並列を指すものではない．

本論文では，第 4 章で述べる手法の対象としてスーパースカラや VLIW を，第 5 章で述べる手法の対象としてデータフロー以外のものでレジスタ数の少ない環境を用いるものとする．

第3章 本論文が用いるデータ表現

以降の章では、いくつかのプログラム最適化手法について述べる。それらすべての手法は、プログラムのソースコードから、ある情報を引き出してコンピュータ上で表現し、その表現に対して決められた処理を行う。我々は、その表現として、プログラム依存グラフ (PDG)[1] の拡張表現と、干渉グラフ [2][3] の拡張表現を用いている。

本章では、我々が用いるプログラムの情報と、それらの表現方法について説明する。

3.1 プログラムの情報

コンパイラが静的に得られるプログラムの情報はいろいろあるが、我々が用いるプログラムの情報は、主に以下のものである。

- 命令間の依存関係
- 仮想レジスタ間の干渉度

それぞれについて詳しく述べる。

3.1.1 命令間の依存関係

プログラム中の命令間には、ある命令がある命令より後に実行されなければ、間違った結果になるという順序関係を作る依存関係が存在する。このような命令の実行順序に関する主な依存関係は以下の通りである。

- 真依存 (true dependency)
- 逆依存 (anti-dependency)
- 出力依存 (output dependency)
- 制御依存 (control dependency)

これらの依存のうち、特に逆依存と出力依存は、名前依存 (name dependency) の一種であり、SSA 形式 (Static Single Assignment Form)[4] などに変換することで、取り除くことが可能である。制御依存については、分岐予測を行ったり、投機的に実行したりすることで緩和することが可能である。

3.1.2 仮想レジスタ間の干渉度

まず、ここでいう仮想レジスタとは、中間言語で値の読み書きが行われる変数の役割を持ち、実レジスタに割付けられるものと定義する。このように定義すると、中間言語内で用いられる仮想レジスタには、他の仮想レジスタと同時に有効な値を保持している区間が存在する。この状態のことを2つの仮想レジスタは干渉しているといい、また、干渉している仮想レジスタ同士は同じ物理レジスタに割付けすることができない。また、干渉している区間での仮想レジスタの使用回数などにより、メモリに仮想レジスタを退避した場合のコストは異なる。このコストのことを干渉度と呼んでいるが、この干渉度を表すためには、干渉があるかないかの1ビットの情報では表せない。また、仮想レジスタの組み合わせごとに干渉度が異なるため、1つの仮想レジスタに対して複数の干渉度が対応する。

また、干渉度は、干渉している区間が同じ長さであっても、仮想レジスタによつての使用頻度が異なるため、最初にどちらの仮想レジスタを物理レジスタに割付けたかによって異なる値になる。例えば、aとbを仮想レジスタとした場合、最初にaをレジスタに割付けた場合と、最初にbをレジスタに割付けた場合では、干渉度が異なる。そこで、本論文では、前者をaからbへの干渉度、後者をbからaへの干渉度と呼ぶ。

3.2 プログラム情報の表現

本手法では、プログラムの情報を見るために、プログラムをグラフ構造で表現している。4章のアルゴリズムが必要とする情報は、最内ループに関する命令数、実行サイクル数、命令間の依存関係である。そこで、4章で用いられるグラフ構造には、これら3つの情報が表せるだけでなく、最内ループ内の分岐、つまり制御依存が表現できる必要がある。また、5章のアルゴリズムが必要とする情報は、各仮想レジスタ間の干渉度である。そこで、5章で用いられるグラフ構造には、この2つの干渉度の違いを表現できる必要がある。

以下では、名前依存を解決するため表現としてSSA形式について説明する。また、上記の条件を満たしたグラフとして、PDGを拡張したGPDG (Guarded PDG)[5]と、干渉グラフを拡張したエッジコスト付干渉グラフについて、それぞれ説明する。

3.2.1 SSA形式

プログラム中に現れるすべての変数名について、代入が1回しか行われれないという形式である。名前依存を取り除くことが可能であるため、さまざまな最適化手法と相性がよく、コンパイラの間言語の段階で、SSA形式が採用されていることが多い。SSA形式への変換方法は以下の通りである。

1. すべての変数について、代入を1回だけに変形する。同一変数 (例えば a) に複数回代入が行われている場合には、リネーミング (例えば a_1, a_2, \dots) を行うことで代入先を変更する。
2. 上記の変更により、分岐の合流点などで、通った経路によって変数名が異なる場合には、経路によって返り値が異なる関数 [4] を挿入する。この関数の返り値を新たな変数名に代入し、それ以降の命令で使用する。

SSA形式への変換例を図3.1に示す。

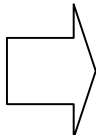
<pre> N = 10000; i = 0; while (i < N) { b = B[i]; tmp1 = y+2; if (b < tmp1){ b = b+1; } else { b = tmp1; } tmp2 = b*12; B[i+1] = tmp2; i = i+1; } </pre>	<p>SSA</p> 	<pre> N = 10000; i0 = 0; while (i1 = ϕ(i0, i2), i1 < N) { b0 = B[i1]; tmp1 = y+2; if (b0 < tmp1){ b1 = b0+1; } else { b2 = tmp1; } b3 = ϕ(b1, b2); tmp2 = b3*12; B[i1+1] = tmp2; i2 = i1+1; } </pre>
--	--	--

図 3.1: SSA 変換の例

例えば，この図において if-else ブロック中の変数 b への代入は，それぞれ b_1, b_2 に置き換えられ，分岐の合流点に関数が挿入されている．この (b_1, b_2) の返り値は，制御が then ブロックから来た場合には b_1 ，else ブロックから来た場合には b_2 となる．

3.2.2 GPDG

PDG は，命令を表すノードと，命令間の依存関係を表す有向エッジから構成されるグラフである．しかし，PDG では制御依存を表すことができない．4章で用いられるグラ

フ構造では，これらに加えて制御依存を表現する必要があるため，ある命令が実行されるかどうかを表すガード (Guard) をノードに追加している．このガードを用いることによって，制御依存関係を真依存関係と同様に，ガードを定義する命令と使用する命令の関係を見ることができる．これによって，最内ループ内に分岐構造があっても，命令間の依存関係などの情報を得ることが可能となる．さらに，グラフの始端と終端を表すため，START ノードと END ノードを導入している．最内ループ内で定義されない変数を使用する命令は START ノードと，最内ループ内で使用されない変数を定義する命令は END ノードと，それぞれ依存関係を持つようにする．

3.2.3 GPDG の例

GPDG の例を図 3.2 に示す．

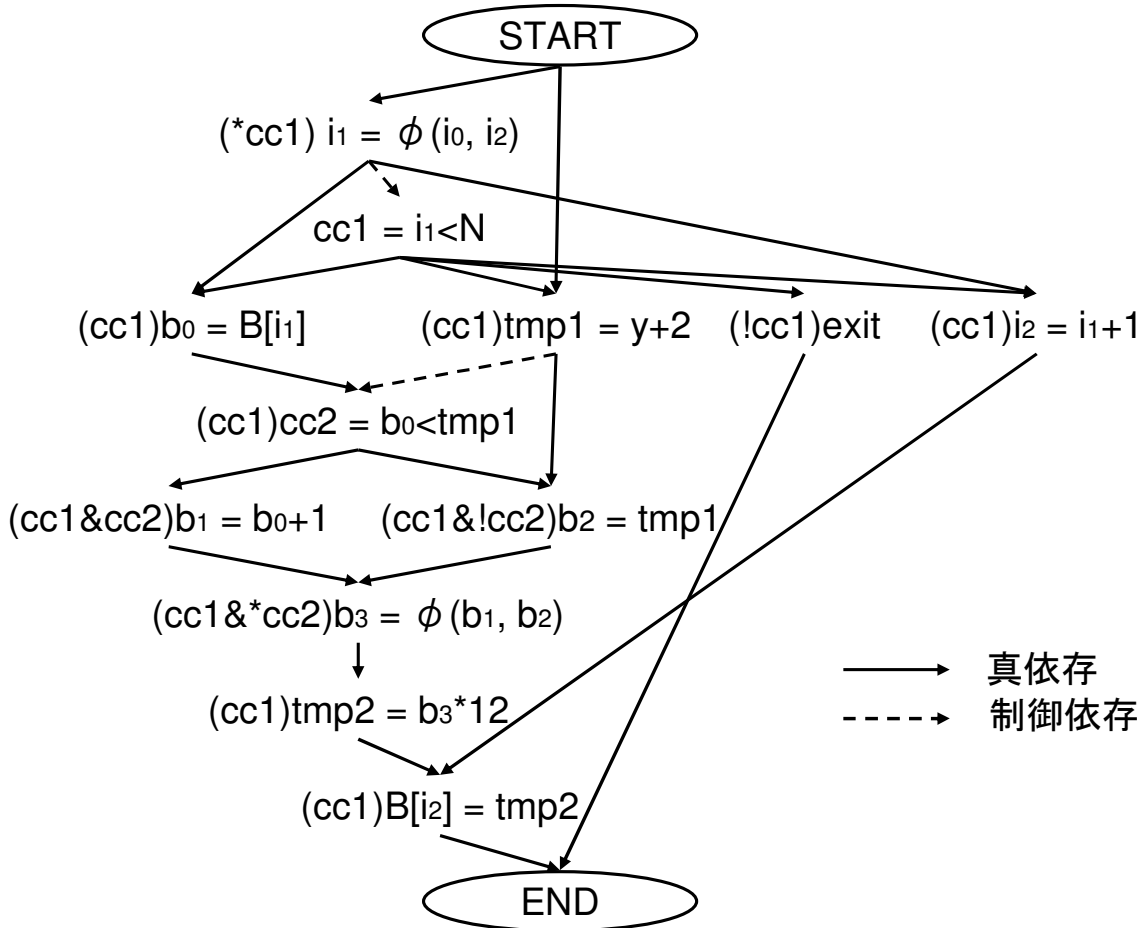


図 3.2: GPDG の例

命令の前に付加されている括弧で囲まれた部分がガードを表している．例えば， (cc) が付加された命令は，条件レジスタ cc が真ならば，命令の実行結果がレジスタに書き込まれる．逆に， $(!cc)$ が付加された命令は，条件レジスタ cc が偽ならば，命令の実行結果がレジスタに書き込まれる．また， $(*cc)$ が付加された命令は，SSA 形式に変換したことによって挿入された関数が返す値を決定づけるのに使用されることを表す．このようにして，PDG 上では表すことができなかった制御依存を表現可能にしている．

3.2.4 エッジコスト付干渉グラフ

干渉グラフとは、仮想レジスタを表すノードと、仮想レジスタ間の干渉を無向エッジから構成されるグラフである。しかし、このグラフでは各仮想レジスタ間に干渉があるかないかは表現されているが、どの程度干渉しているかは表現できない。5章で用いられるグラフ構造では、各仮想レジスタがどの程度干渉しているかという干渉度を表現する必要があるため、エッジを有向かつ双方向にし、各エッジが干渉度を保持するようにしている。これによって、干渉グラフ上において、1ビットという2つの値では表すことができない干渉度を表すことを可能にしている。また、1つの仮想レジスタは1つのコストしか持てなかったが、これによって干渉している各仮想レジスタとの干渉度、つまり複数の干渉度を1つの仮想レジスタが持つことも可能になっている。

3.2.5 エッジコスト付干渉グラフの例

エッジコスト付干渉グラフの例を図 3.3 に示す。

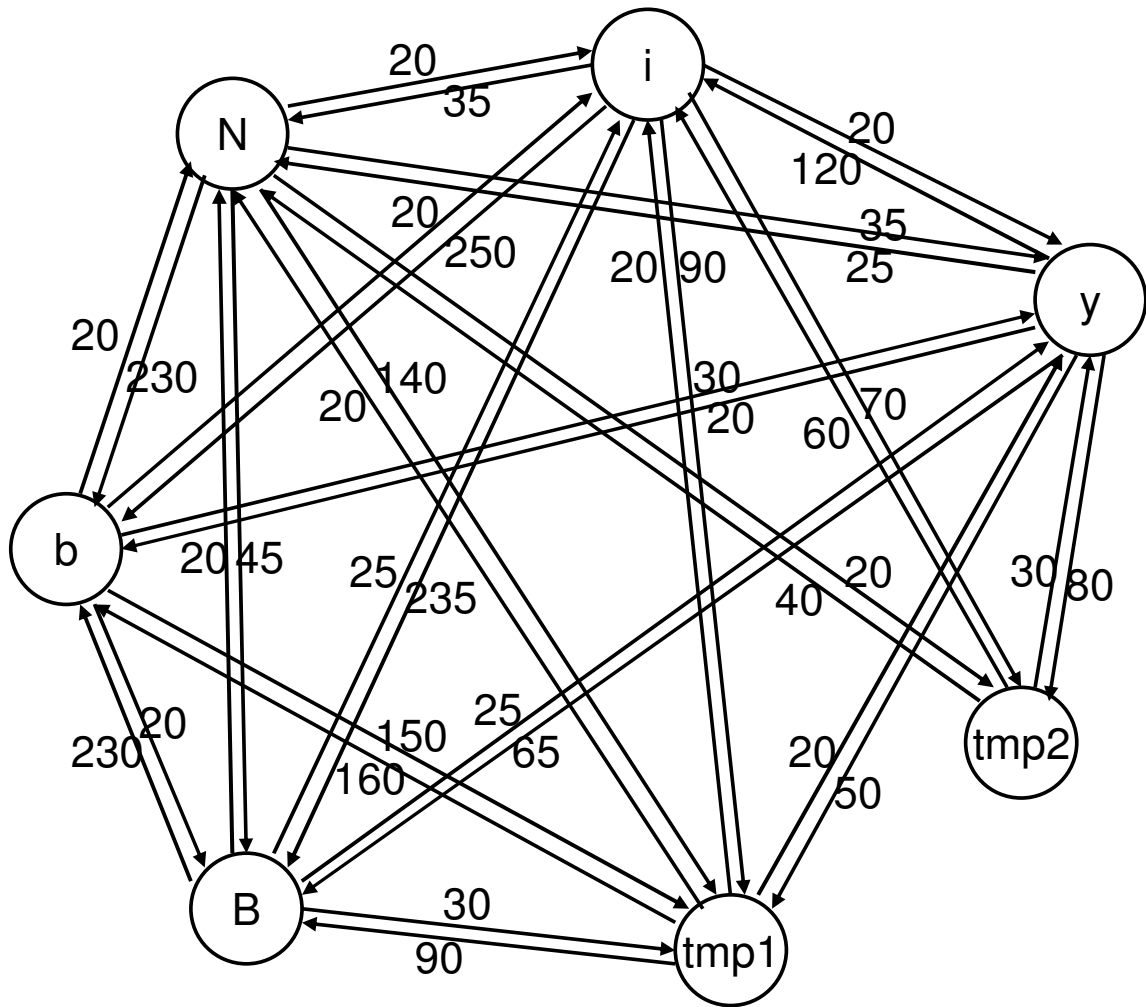


図 3.3: エッジコスト付干渉グラフの例

ノードが仮想レジスタ，ノード間を接続するエッジが干渉を表している点は，干渉グラフと同様である．異なる点は，エッジが有向エッジであり双方向になっていることと，各エッジにコストを表す整数値が付いている点である．例えば，ノード N からノード B に向かうエッジが表すコストは，ノード N の表す仮想レジスタとノード B の表す仮想レジスタで同じレジスタを共有した場合に，ノード N の表す仮想レジスタを優先的にレジス

タに割付けた場合のスピルコストを表している。つまり、ノード N の表す仮想レジスタからノード B の表す仮想レジスタへの干渉度を表す。同様に、逆向きのノード B からノード N に向かうエッジが表すコストは、ノード B の表す仮想レジスタを優先的にレジスタに割付けた場合のスピルコストを表している。つまり、ノード B の表す仮想レジスタからノード N の表す仮想レジスタへの干渉度を表す。

第4章 プロセッサ並列度を考慮したレジスタ割付け技法

4.1 まえがき

現在主流となっているアーキテクチャでは，物理レジスタへのアクセスコストに比べて，メモリへのアクセスコストが高い．そのため，メモリへのアクセスが頻繁に起こると，プログラムの実行効率は大きく低下してしまう．そこで，プログラムを高速実行するためには，コンパイラによる物理レジスタの利用に関する最適化が必須となってくる．

パイプラインが1つの場合には，メモリへのアクセス回数を減らすには，単純に物理レジスタの再利用性を高めればよく，レジスタ割付けの結果が最適解であるならば，メモリへのアクセス回数は最小化できる．しかし，パイプラインが複数ある場合には，メモリへのアクセス回数を減らすために，物理レジスタの再利用性を高めると，命令を入れ替えを阻害する依存が発生しやすくなる．そのため，これまでのレジスタ割付け手法で最適解に近い結果であっても，命令の入れ替えを充分に行うことができず，プログラムの命令レベル並列性 (ILP:Instruction Level Parallelism) を引き出すことが困難な場合がある．ILPが充分に引き出せない場合には，メモリへのアクセス回数を少し増やした方が高速に実行できる場合もある．これは，メモリへアクセスを行う演算が他の整数演算などと並列に実行でき，メモリへのアクセス時間が隠蔽されるからである．

そこで，メモリへのアクセス回数を減らしつつ充分な ILP も引き出すためには，スピルコードを少なくするとともに，並列実行についても考慮するレジスタ割付け手法が必要である．つまり，コードスケジューリングの動作を考慮して命令を入れ替えられる余地を

残しながら、物理レジスタ数制約を満たすようなレジスタ割付け手法が必要ということである。コードスケジューリングとは、同時実行できる命令を見つけて ILP を高めることによって実行効率を向上させ、プログラム的高速実行を可能にする最適化である。一般的に、ILP を高めるとプログラム中で用いられる変数の値である仮想レジスタの同時刻に生存する数は増える傾向がある。この動作は、数に限りのある物理レジスタに仮想レジスタをできるだけ割付けるために、一般的に ILP を減らす動作を行うレジスタ割付けとは、逆の動作となっている。このレジスタ割付けとコードスケジューリングという2つの最適化は、コンパイラにおいてどちらも不可欠な最適化であるため、同時に最適化を行うかいずれかの最適化を先に行うことになる。しかし、同時に2つの最適化を行うことは非常に困難であるため、現時点で提案されている手法は存在しない。また、レジスタ割付けとコードスケジューリングのどちらを先に行う場合には、前述のようにお互いに影響しあう問題が存在する。そのため、物理レジスタ数の制約を満たしつつ十分な ILP を引き出すためには、お互いの協調動作を考慮する手法である必要がある。

本章では、この協調動作を考慮したレジスタ割付け手法について提案する。

4.2 本手法の概要

本手法は、大きく分けて2つのステージに分けることができる。1つは、コード生成の対象とするアーキテクチャの物理レジスタ数制約を満たすステージであり、もう1つは、機能ユニットの数を考慮しながらコード生成を行うステージである。本手法では、前半の操作を「最大干渉数低減操作」、後半の操作を「プレスケジューリングおよびコード生成」と呼ぶ。それぞれについて以下で説明する。

4.2.1 最大干渉数低減操作

ここでは、この最大干渉数をコード生成の対象とするアーキテクチャの物理レジスタ数以下に低減する操作を行う。最大干渉数とは、プログラムの実行時系列におけるすべての要素において、同時に生存している仮想レジスタの数のうち最大のものを指す。つまり、この最大干渉数がコード生成の対象とするアーキテクチャの物理レジスタ数以下ならば、機能ユニットの数が無限にあるときにレジスタ割付けが可能であることを表している。機能ユニットの数が無限にあるときという条件が付く理由は、ある時系列要素を t 、1つ先の要素を $t+1$ としたとき、 $t+1$ における干渉数は、 t と $t+1$ の間で行われる命令がすべて並列に実行された場合に生存している仮想レジスタ数を表しているからである。この t と $t+1$ の間で行われる命令が、コード生成の対象とするアーキテクチャの機能ユニットを上回っている場合には物理レジスタが足りなくなる可能性もある。このため、最大干渉数がコード生成の対象とするアーキテクチャの物理レジスタ数以下という条件だけでは、レジスタ割付けが可能である保証にはならない。

最大干渉数低減操作に対する入力は、レジスタ生存グラフ [5] というグラフを用いる。このレジスタ生存グラフは、仮想レジスタをノードで表し、その仮想レジスタの生成と使用の関係を、ノード間を有向エッジで接続することで表現したグラフである。また、このグラフでは、プログラムの実行時系列におけるすべての要素において、同時に生存している仮想レジスタを表すノードは、1本の等時刻線と呼ばれる線分で結ばれる。この等時刻線によって結ばれたノードの数は、その等時刻線が表すプログラムの実行時系列における要素において同時に生存している仮想レジスタ数を表していて、これを干渉数と呼ぶ。この干渉数の中で、すべての等時刻線において最大のものが、先ほど定義した最大干渉数である。また、レジスタ生存グラフ上の各ノードは、自由度と呼ばれるパラメータを持っている。この自由度とは、プログラムの実行時系列の要素数を増やさずに、そのノードが表す仮想レジスタを使用する演算を、どれほど遅らせることが可能かという指標である。

このレジスタ生存グラフの一例を図 4.1 に示す。

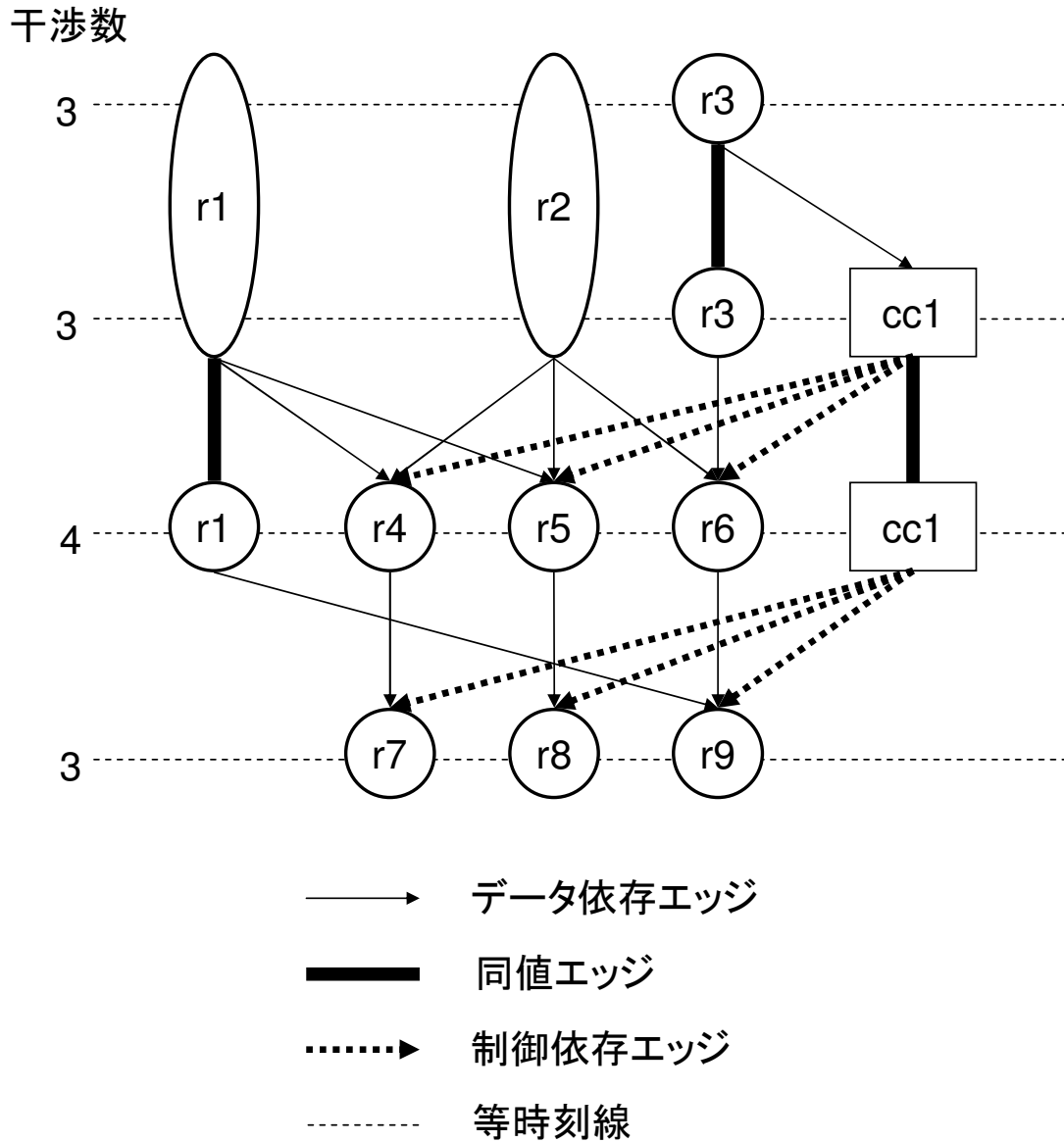


図 4.1: レジスタ生存グラフの例

この例では、円や楕円で描かれているものが仮想レジスタであり、その生成と使用の関係を実線の有向エッジで表している。同値エッジについては本来なくてもよいものだが、仮想レジスタの使用の関係を表すエッジがノードの途中で始まる場合でも、矢印の出所を

見やすくするために使用している。また、四角で描かれているものがガード情報であり、制御依存の関係を点線の有向エッジで表している。また、このグラフで真横に伸びる線分が等時刻線であり、プログラムの各実行時系列を表している。そして、この等時刻線が貫く仮想レジスタの数が、干渉数となる。ガード情報については、条件レジスタと呼ばれる特殊な物理レジスタに割付けるため、干渉数には含めないことにする。

このグラフに対して、コード生成の対象とするアーキテクチャの物理レジスタ数より干渉数が高いプログラムの時系列におけるすべての要素に対して、演算に対して物理レジスタ数制約を満たすための依存を付加するか、仮想レジスタに対してスピル操作を行うことで最大干渉数を下げる操作を行う。最大干渉数低減操作からの出力は、最大干渉数がコード生成の対象とするアーキテクチャの物理レジスタ数以下になっているレジスタ生存グラフである。

4.2.2 プレスケジューリングおよびコード生成

最大干渉数低減操作では、コード生成の対象とするアーキテクチャに存在する機能ユニットの個数が無限であるという仮定があった。しかし、実際のアーキテクチャでは、機能ユニットの個数は無限ではなく数に限りがある。この仮定と現実の差を埋めるために、プレスケジューリングを行う。

ここで、機能ユニットの数が無限という仮定と、機能ユニットの数が有限という現実の差が生む物理レジスタ不足の例を図 4.2 に挙げる。

物理レジスタ数:2

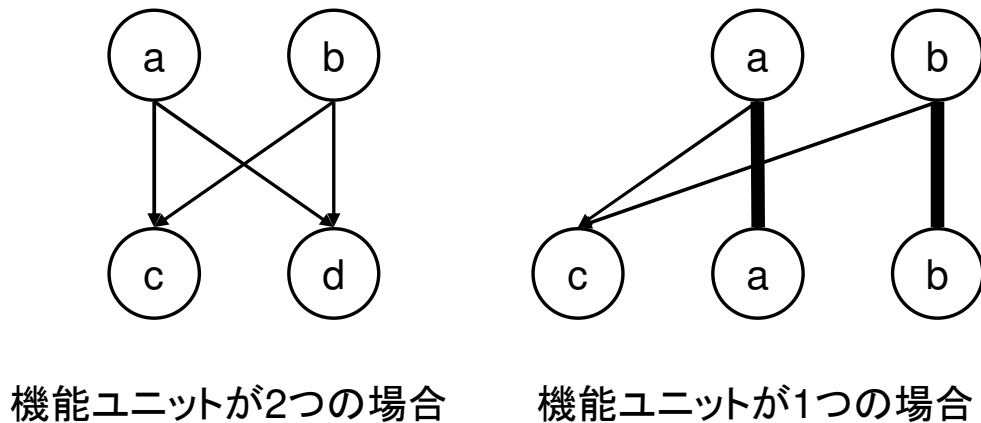


図 4.2: 機能ユニット数の違いによる物理レジスタ不足の例

この例では、物理レジスタ数2で a, b という2つの値があり、同じ時系列で $c = a + b, d = a - b$ という2つの命令が実行できる状態のとき、機能ユニットが2つの場合と1つの場合で、仮想レジスタの状態がどのように異なるかを表している。機能ユニットが2つの場合には、この2つの演算を並列に実行して、次の時系列の物理レジスタには c, d という2つの仮想レジスタが入っていることになり、物理レジスタ不足は起こらない。これは、図4.2の左側のレジスタ生存グラフで表せる。しかし、機能ユニットが1つの場合には、 $c = a + b$ と $d = a - b$ のどちらの演算を実行したとしても、次の時系列では a, b と c または d という3つの仮想レジスタが必要であり、物理レジスタが1つ不足する。これは、図4.2の右側のレジスタ生存グラフで表せる。このように、対象とするアーキテクチャの機能ユニットの数を考えずにコードを生成してしまうと、予想外のところでメモリとのやり取りが発生し、想定していたより性能を落としてしまう可能性がある。最悪の場合には、まだ使用する物理レジスタの値を演算結果で上書きしてしまい、以後の計算結果が間違える可能

性もある。プレスケジューリングでは、このような事例が起こらないかを事前にチェックする。

プレスケジューリングに対する入力は、機能ユニットの個数が無限であるという仮定の上で、コード生成の対象とするアーキテクチャの物理レジスタ数制約を満たしているレジスタ生存グラフである。つまり、物理レジスタ数制約を満たすように、最大干渉数低減操作を施したレジスタ生存グラフである。

このグラフを用いて、プログラムの時系列におけるすべての要素に対して、コード生成の対象とするアーキテクチャに存在する機能ユニットの個数で起こりうる演算の組をすべて作り、それぞれの演算組の実行前と実行後の物理レジスタの状態をシミュレートする。ただし、組み合わせ爆発を起こしてしまうため、ある時系列で演算の組を作成したときに、その時系列で選ばれなかった演算を次の時系列の演算に含めたりはしない。実行後に物理レジスタが不足する状態になることを検出した演算の組については、その演算の組み合わせが発生しないように、演算に対して機能ユニット数制約を満たすための依存を付加するか、仮想レジスタに対してスピル操作を行う。プレスケジューリングからの出力は、コード生成の対象とするアーキテクチャにある機能ユニットの個数で、物理レジスタ数制約も満たしているレジスタ生存グラフである。

4.3 本手法の流れ

ここでは、サンプルプログラムから、コード生成が行われるまでの本手法の流れを示す。本手法は主に、プログラムの最内ループに対して最適化を行う。なぜならば、最内ループはプログラム中で最も多く実行されることが期待され、最適化の効果が最も大きい部分だからである。本手法の全体の流れを、図 4.3 に示す。

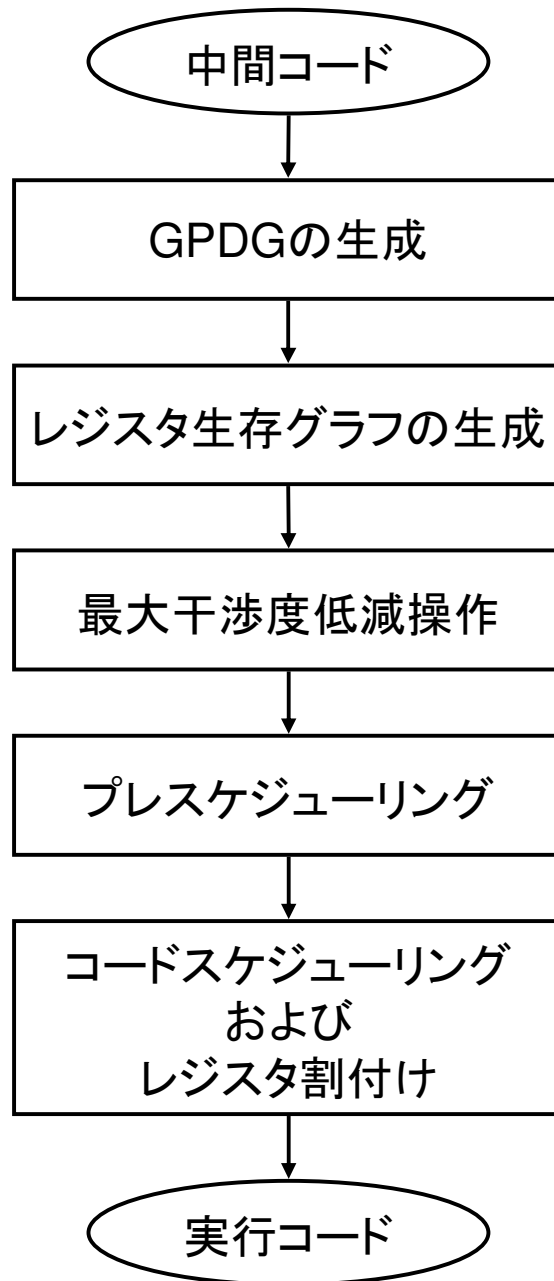


図 4.3: 本手法の流れ

本手法の処理は、このようなステップに分けることができる。まず、中間コードから最内ループを検出し、最内ループに属する命令群について GPDG を作成する。そして、その GPDG から仮想レジスタの定義と使用の関係を表すためにレジスタ生存グラフを作成する。次に、物理レジスタ数制約を満たすために最大干渉数低減操作を施し、プレスケジューリングを行うことで、実際に機能ユニットに割付ける段階での保証とする。最後に、レジスタ生存グラフへ施した変更を元の GPDG にも反映させる。この変更した GPDG を用いて依存が発生しないようにレジスタ割付け・コードスケジューリングを行う。

それぞれについて、以下で詳しく説明する。

4.3.1 GPDG の作成

CFG (制御フローグラフ) の構築

まず、プログラムの実行の流れを表すために、CFG (Control Flow Graph) を構築する。

CFG を作成するには、プログラム内の基本ブロック (Basic Block) を検出する必要がある。ここで、基本ブロックに属する最初の命令を、その基本ブロックのリーダーと定義する。一般的に、リーダーになりうる命令は、分岐命令のジャンプ先の命令、分岐命令の次の命令と、関数の最初の命令という 3 種類存在する。

リーダーから次のリーダーの直前の命令まで、または、関数の最後の命令までの命令列が、1つの基本ブロックとなる。CFG を構築するには、検出したすべての基本ブロックをノードで表し、ノード間に有向エッジを張る必要がある。有向エッジを張り方を以下に示す。

1. 無条件ジャンプが属する基本ブロックを表すノードから、そのジャンプ先の命令が属する基本ブロックを表すノードへ向けて有向エッジを張る。
2. 無条件ジャンプも分岐命令も属さない基本ブロックを表すノードから、その基本ブロックに属する最終命令の直後の命令を持つ基本ブロックを表すノードへ向けて有

向エッジを張る。

3. 分岐命令が属する基本ブロックを表すノードから，その分岐命令のジャンプ先の命令が属する基本ブロックを表すノードへ向けて有向エッジを張る．その有向エッジには分岐成立を示すラベルを付ける．
4. 分岐命令が属する基本ブロックを表すノードから，その分岐命令の次の命令が属する基本ブロックを表すノードへ向けて有向エッジを張る．その有向エッジには分岐不成立を示すラベルを付ける．

支配木 (dominance tree) の作成

ここでは，基本ブロック間の支配関係を表すために，CFG から支配木を構築する．ここで作成された支配木の情報は，後ほど行われる最内ループ (inner-most loop) の発見，SSA (Static Single Assignment) 変換に用いられる．

ここでいう支配関係とは，CFG 上で開始ノードからあるノード A までのすべてのパスが，必ずノード B を通らなければならない場合の，A と B との関係のことを指す．この場合は，B は A を支配 (dominate) するという．この定義より，開始ノードを除いた，あるノード A を支配するノードは，開始ノードと自分自身 A の少なくとも 2 つ存在することになる．この支配するノードの集合から自分自身を除いた集合を，厳密支配 (strictly dominate) するノードの集合と呼ぶ．また，ノード A を異なる 2 つのノード B とノード C が厳密支配していて，さらにノード C をノード B が厳密支配するようなノード B が存在しないとき，ノード C はノード A を直接支配 (immediately dominate) するといい，C を A の直接支配ノードと呼ぶ．この直接支配ノードは，開始ノードには存在せず，それ以外のノードには必ず 1 つだけ存在する．この直接支配ノードの関係を，ノード同士を無向エッジで結ぶことで表現していくことで，支配木は作成できる．

この支配木を構築するために，開始ノードを除いた各ノードに対する直接支配ノードの求め方を記す．まず，開始ノードを含めた各ノードに対して，そのノードを支配するノード

ド集合を作成する必要がある。ここで、 $D[n]$ をノード n を支配するノード集合、 $pred[n]$ をノード n への有向エッジを持つノード集合、つまり先行ノード集合とすると、 $D[n]$ は、

$$D[n] = n \cup \left(\bigcap_{p \in pred[n]} D[p] \right)$$

で求めることが可能である。次に、得られた $D[n]$ の中から、開始ノードを除いた各ノード n について、前述した条件を満たす直接支配ノードを探す。

基本ブロックを表すノードを用意し、直接支配の関係を持つノード同士に無向エッジを張ることで、支配木を構築する。

最内ループ (inner-most loop) の発見

本手法は、最内ループのみに適用する。しかし、最内ループ内に関数呼び出しの命令などが存在する場合には、関数部分の実行時間を正確に計算できないため、このままでは最適化できない。本来ならば、関数呼び出しがループ内に存在する場合、関数の本体も繰り返して実行されるため、ループ内の他の命令とともに最適化されるべきである。

そのため本手法では、最内ループ中の関数呼び出しに関しては、再帰呼び出し構造になっている関数以外は、CFG 上にてインライン展開を行っている。また、最内ループ以外の部分に関しては、既存のコンパイラに任せることにする。

まず、CFG からプログラム中に存在するすべてのループを検出する。ループとは以下の条件を満たすノードの集合である。

1. 始点 (ループヘッダ) がある
2. 集合への入り口は始点のみである
3. 集合に属するノードからは、その集合に属するどのノードへも到達可能である

特に3つ目の条件を満たすためには、ループヘッダへの後向きのエッジ (バックエッジ) を持っている必要がある。したがって、まず CFG においてバックエッジを持つノードを見

つけ，そこからループに属する集合を作成し，上記3つの条件に合うものに絞ればよい．この集合は，ループヘッダが支配するノード集合の部分集合になる．

次に，各ループで互いに共通の要素を持っているかどうかを確認する．これは，ループの中にあるループを検出するために必要である．複数のループが共通の要素を持つのは，次の2つの場合だけである．

1. 共通のループヘッダを持つ

同一ループヘッダへ，2ヶ所以上から分岐命令などでジャンプしてくる状態である．最内ループを検出する場合には，共通のループヘッダを持つループをすべて統合することで，1つのループとして扱う．ただし，いずれかのループが最内ループの候補外になった場合には，統合した他のすべてのループも候補から外す必要がある．

2. ループを内包している

ループが入れ子になっている状態である．最内ループを検出する場合には，外側のループを最内ループの候補から外す．この際に，外側のループヘッダを表すノードから支配木をたどり，内側のループヘッダが存在しないかを判定し，最内かどうかを判断する．

上記の操作をすべてのループについて終えた時点で，他のどのループとも共通要素を持たないループが最内ループとなる．

SSA 形式への変換

本手法では，仮想レジスタを再利用したために命令の入れ替えによって逆依存が発生することを防ぐため，最内ループ中に出現するすべての変数に対して，SSA 変換を行う．SSA 変換とは，プログラム中で各変数の定義が1ヶ所で行われなければならないように，かつプログラムの意味を変えないように，変数名を書き換える操作を施すことである．SSA 変換

を施した状態を SSA 形式と呼び、この状態はコード移動や部分冗長性削除、定数伝播などに適している。その手順は次の通りである。

まず、各ノードに対して、dominance frontier[4] という概念を利用して、CFG 上で関数を挿入する場所を探す必要がある。ここで、あるノードの dominance frontier とは、そのノードの支配関係が及ばなくなる境界のノードの集合を指す。つまり、ノード X のある先行ノードへの経路はすべてノード Y を通るが、ノード X へはノード Y を通らないパスが存在するとき、「 X は Y の dominance frontier である」という。また、関数とは、分岐による異なるパス上で同じ変数名で異なる値が定義される場合に、その分岐合流点でその変数名で定義された正しい値を得るための関数である。

CFG において、 $succ[n]$ をノード n の後続ノード集合、 $idom[n]$ をノード n の直接支配ノード、 $child[n]$ を $idom[k] = n$ となるノード k の集合と定義すると、次のようにノード x の dominance frontier の集合 ($DF[x]$) を定義できる。

$$DF[x] = \{y \in succ[x] \mid idom[y] \neq x\} \cup \bigcup_{z \in child[x]} \{y \in DF[z] \mid idom[y] \neq x\}$$

これを基にして、あるノードの集合 S の dominance frontier の集合 ($DF(S)$) を、次のように定義する。

$$DF(S) = \bigcup_{x \in S} DF[x]$$

さらに、 $DF^1(S) = DF(S)$ 、 $DF^{i+1}(S) = DF(S \cup DF^i(S))$ として、dominance frontier の閉包 $DF^+(S)$ を次のように定義する。

$$DF^+(S) = \lim_{i \rightarrow \infty} DF^i(S)$$

各変数に対して、CFG 上で定義が含まれる基本ブロック S を求め、 $DF^+(S)$ を計算することで、関数を挿入するすべての位置を見つけ出すことができる。関数の引数については、本手法では2つとしている。なぜなら、引数が3つ以上ある場合には、コード生成時にどの順番で引数同士を計算すると効率がよいかなどといった、性能を考慮する必

要性が出てくるからである。また、本手法は最内ループのみに対象を絞っているため、あるノードの dominance frontier は、そのノードの後続ノードに必ず存在する。このことから、プログラムの実行方向とは逆向きの探索をする必要がないので、dominance frontier の発見を高速化できる。

ただし、ここで述べたものは最小 SSA と呼ばれ、本来必要のない無駄な 関数も挿入されてしまう。たとえば、後続ノードで使用されない変数の 関数などがこれにあたる。プログラムの性能を上げるためには、無用命令の削除などを行い、必要最低限な 関数以外を除く必要がある。

最後に、CFG 上で、各変数の定義と参照に対してリネーミングを行う。例えば、変数 x について、 x_1, x_2, x_3, \dots のように変数名を書き換える。同じ変数名が使われる範囲は、変数の定義命令からその変数の次の定義命令まで、もしくは、その変数の 関数までである。本手法では最内ループのみに対象を絞っていることから、dominance frontier を求める時と同様に、プログラムの実行方向とは逆向きの探索を行わないことで高速化を図れる。

例として、図 4.4 の (a) にサンプルコードを、(b) にその SSA 形式を示す。本手法の説明には、今後このサンプルコードを用いる。

```

if (r2!=0) {
    r6=r4+r5
    r7=r4-r5
    r8=r5+1
    r9=r3>>2
    store(r9,r6)
    store(r9,r7)
    r9=r2-1
} else {
    r6=Load(r4)
    r7=r6<<1
    store(r7,r3)
}

```

(a) サンプルコード

```

cc1=r2&0
(!cc1)r6=r4+r5
(!cc1)r7=r4-r5
(!cc1)r8=r3>>2
(!cc1)r9=r5+1
(!cc1)store(r8,r6)
(!cc1)store(r8,r7)
(!cc1)r10=r2-1
(cc1)r11=Load(r4)
(cc1)r12=r11<<1
(cc1)store(r12,r3)

```

(b) SSA変換,ガード付加後

図 4.4: サンプルコード

CDG (制御依存グラフ) の構築

CDG (Control Dependence Graph) においては、各基本ブロックをノードで表し、基本ブロック間の分岐依存関係を無向エッジで表す。この無向エッジには、分岐条件の成立不成立を表すガードというものを付加する。このCDGを用いることによって、基本ブロック間の制御依存関係が把握しやすくなる。ここで、ノードYがノードXに制御依存するという関係は、ノードXからある辺を辿った場合には、その後必ずノードYを通らなければならないが、それとは別の辺を辿った場合には、ノードYを通らない経路があるという関係のことである。このCDGでは、あるノードをルートとしたとき、ルートからルー

トを頂点とした部分グラフのすべてのノードに対して制御依存があることを示す。

それでは、CFG から CDG の生成方法について述べるために、まずポスト支配 (post dominate) について説明する。CFG において、ノード A から END ノードまで行くすべてのパスが、ノード B を必ず通過しなければならない場合、ノード B はノード A をポスト支配するという。この定義より、あるノードは自分自身を必ずポスト支配していることになる。また、ノード A とノード B が異なるノードで、かつ、ノード B にポスト支配されていてノード A をポスト支配するようなノードがノード B 以外に存在しない場合、ノード B はノード A を直接ポスト支配 (immediately post dominate) するという。支配木同様、ポスト支配木 (post dominance tree) では、各基本ブロックをノードで表し、直接ポスト支配関係を無向エッジでノード同士を結ぶことで表す。

ポスト支配関係は、CFG の有向エッジがすべて逆方向になったグラフでの支配関係として捉えることができる。そこで、CFG に存在するすべての有向エッジの方向を逆にしたグラフを作成し、そのグラフに対して支配木を構築するアルゴリズムを適用すれば、ポスト支配木を得られる。

CFG の有向エッジの中で、その終点が始点をポスト支配していないエッジのことを分岐エッジという。CFG の各分岐エッジの始点ノード A と終点ノード B に対して、ポスト支配木上で、ノード A とノード B の共通の祖先であるノード C から、ノード B までのパスを探し出す。このパス上に存在するノード C を除いたすべてのノードは、ノード A に制御依存する。ノード A からノード A に制御依存するノードすべてにエッジを張る。

PDG の構築

PDG では、各命令をノードで表し、データ依存を有向エッジで表す。PDG を構築する手順は以下の通りである。

1. 最内ループ内に命令全てに対して、以下の操作を繰り返す。
 - (a) 最内ループ内の命令を、無条件ジャンプ命令とラベル命令を除いて選択する。

- (b) 選択した命令を表すノードを作成し，PDG に追加する．
 - (c) 前ステップで追加した命令と，PDG に以前追加した命令との間に，依存関係がある場合には，PDG に以前追加した命令を表すノードから，追加する命令を表すノードに向けて有向エッジを張る．
2. PDG の先頭と末尾に，始点を表す START ノードと，終点を表す END ノードを追加する．
 3. START ノードから，最内ループ外で定義されたデータを参照する命令を表すノードへ向けて有向エッジを張り，最内ループ内で参照されないデータを定義する命令を表すノードや store 命令から，END ノードへ向けて有向エッジを張る．

GPDG の構築

前ステップまでに作成した CDG と PDG を組み合わせることによって GPDG を作成する．GPDG を作成する手順は次の通りである．

1. CDG において，どのノードにも制御されていないノードを選ぶ．複数の候補が存在する場合，その中から任意の 1 つを選ぶ．
2. 選択されたノードが表す基本ブロックに分岐命令がある場合，PDG に対して次の操作を行う．
 - (a) 分岐命令を表すノードを，ガードを定義するノードに変換する．
 - (b) CDG において，その分岐命令を含む基本ブロックを表すノードが，ルートとなる部分グラフにあるルートを除いたすべてのノードが表す基本ブロックに含まれる命令すべてに対して，PDG で変換したガードを定義するノードから，これらの命令を表すノードに向けて制御依存を表す有向エッジを張る．

(c) CDG において，その分岐命令を含む基本ブロックを表すノードから，有向エッジを張られているノードが表す基本ブロックにある命令すべてに対して，PDG で次のようにガードを生成する．

- まだガードが生成されていない命令には，変換したガードを定義するノードが持つガードと定義したガードの論理積を，その命令の実行条件として追加する．
- 既にガードが生成されている命令には，変換したガードを定義するノードが持つガードと定義したガードの論理積を計算し，その論理積と元のガードとの論理和を，その命令の新しいガードとする．

3. CDG から選択されたノードと，選択されたノードが持つエッジをすべて削除する．

上記の操作を CDG が空になるまで繰り返す．

図 4.4 のサンプルコード (b) から生成した GPDG を，図 4.5 に示す．

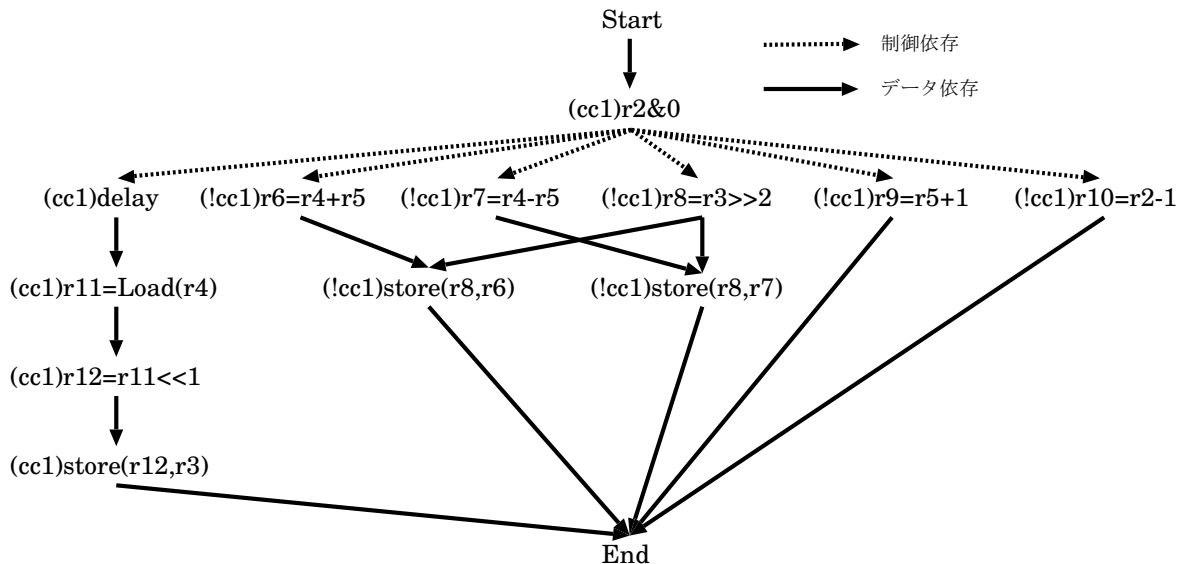


図 4.5: GPDG の生成

このように，命令間の真依存と制御依存のみによる実行順序がわかり，最大限の並列性が抽出されているのが分かる．

4.3.2 レジスタ生存グラフの作成

前ステップで得られた GPDG から，仮想レジスタの定義と使用の関係を示すため，レジスタ生存グラフを作成する．レジスタ生存グラフを作成する手順は次の通りである．

1. レジスタ生存グラフの始点を表すノードとして TOP ノードを，また終点を表すノードとして BOTTOM ノードを作成する．
2. GPDG のすべてのノードについて以下の操作を繰り返す．
 - (a) GPDG から 1 つノードを選ぶ．
 - (b) 選択した GPDG のノードが表す命令が，定義する仮想レジスタを表すノードを作成する．もし，その命令が store 命令だった場合には，store ノードを作成する．
 - (c) 選択した GPDG のノードが表す命令が，使用する仮想レジスタを表すノードから，先ほど作成したノードへデータ依存エッジを張る．もし，使用する仮想レジスタを表すノードが，レジスタ生存グラフ内に存在しない場合，その仮想レジスタを表すノードを新たに作成し，TOP ノードからそのノードへデータ依存エッジを張ってから，データ依存エッジを張る操作を行う．
3. 定義された後，使用されない仮想レジスタを表すノード，つまり他のノードへのデータ依存エッジを持たないノードから，BOTTOM ノードへ向けデータ依存エッジを張る．
4. レジスタ生存グラフの各ノードについて，定義されたサイクルと不必要になるサイクルをそれぞれ求める．

5. 各サイクルごとに，生存している仮想レジスタを表すノードをすべて横切るように等時刻線を引く．
6. 等時刻線が横切ったノードの数を，そのサイクルの干渉数として求める．

図 4.5 の GPDG から生成したレジスタ生存グラフを図 4.6 に示す．

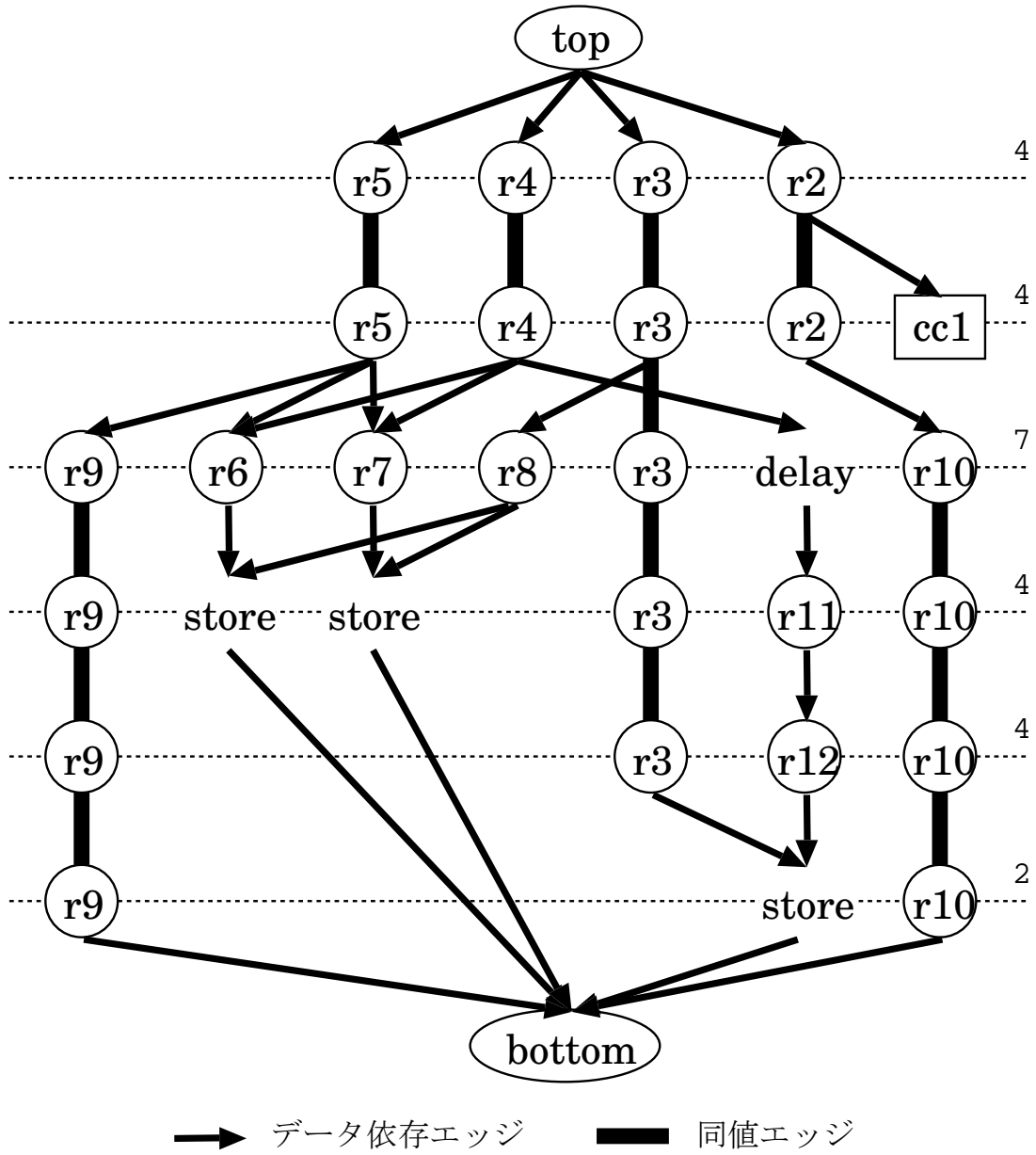


図 4.6: レジスタ生存グラフの生成

4.3.3 最大干渉数低減操作

レジスタ生存グラフ上で、ある時刻での干渉数が実レジスタ数を超えている場合、レジスタ割付け時にレジスタが足りなくなることを示している。そこで、すべての時刻での干渉数を、つまり最大干渉数を実レジスタ数以下に低減する必要がある。

クリティカルパス長の計算

クリティカルパスとは、グラフにおいて始点から終点までの最長パスのことである。レジスタ生存グラフの場合、このクリティカルパス長は、ほぼプログラムの最内ループにかかる実行時間とみることができる。この最大干渉数低減操作では、クリティカルパス長を評価値として、できるだけ長くならないように変形を施す。

レジスタ生存グラフでの、クリティカルパス長の求め方は、レジスタ生存グラフ上で最も深いところにあるノードのサイクル数を求めることと同じである。

自由度の計算

自由度とは、レジスタ生存グラフ上で、クリティカルパス長に影響しないように、仮想レジスタを表すノードの生成を、どの程度遅らせることができるかという指標である。つまり、クリティカルパス上にあるすべての仮想レジスタを表すノードは、自由度が0となる。

自由度の計算を行うには、まず GPDG の各ノードのサイクル数を計算する必要がある。この計算は、GPDG の START ノードから再帰的に行う。GPDG の各ノードのサイクル数を求める手順は次の通りである。

1. Start ノードからのエッジを持つノードのサイクルを 1 にする。
2. サイクル数が決定しているノードからのエッジを持つノードを 1 つ選ぶ。ここでは、サイクル数が決定しているノードを X、選んだノードを Y とする。

3. ノード X のサイクル数に、ノード X が表す命令にかかるサイクル数を加えたものを、ノード Y のサイクル数にする。ただし、ノード Y がすでにサイクル数が決定しているノードの場合は、サイクル数の大きいものを、ノード Y のサイクル数とする。

こうして求めた GPDG の各ノードのサイクル数から、レジスタ生存グラフの各ノードの自由度を求める。この計算は、GPDG の END ノードから再帰的に行う。レジスタ生存グラフの各ノードの自由度を求める手順は次の通りである。

1. End ノードへのエッジを持つノードの自由度を、END ノードのサイクルから、そのノードのサイクル数とそのノードが表す命令にかかるサイクル数を引いた値にする。
2. 自由度が決定しているノードへのエッジを持つノードを 1 つ選ぶ。ここでは、自由度が決定しているノードを X、選んだノードを Y とする。
3. ノード X のサイクル数から、ノード Y のサイクル数とノード Y が表す演算にかかるサイクル数を引いたものを、ノード Y の自由度にする。ただし、ノード Y がすでに自由度が決定しているノードの場合は、自由度の小さいものを、そのノードの自由度とする。

レジスタ生存グラフでの各ノードの自由度は、そのノードが表す仮想レジスタの値を定義する命令を表す GPDG のノードの自由度になる。

干渉数の低減

干渉数を下げる方法は、大きく分けて 2 つある。1 つは、クリティカルパス長に影響が少ない演算の実行を遅らせる方法 (干渉数低減操作 1) である。図 4.7 に干渉数低減操作 1 の適用例を挙げる。

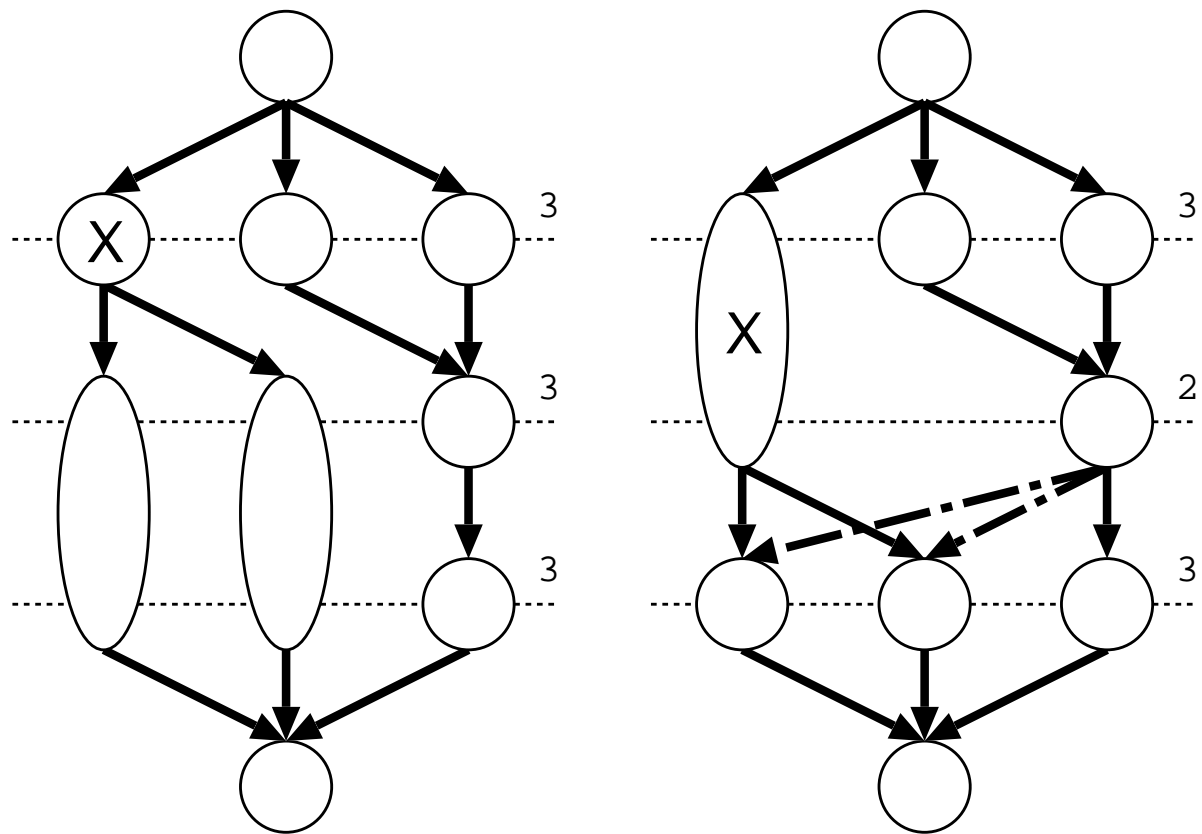


図 4.7: 干渉数低減操作 1 の例

このような例の場合，ノード X が表す仮想レジスタの値を使用する演算を 1 サイクル遅らせて，干渉数を 1 つ下げることができる．この際に，ノード X から生成される仮想レジスタに対して，元の時系列で生成される仮想レジスタと仮想的な依存を付加する．これは，実際には条件がそろっているのだが，条件がそろっていないと見せかけて，演算の実行を遅らせるために行う．

もう 1 つは，スピルコードを挿入することで，すぐには参照されなくてもかまわないデータをメモリに退避する方法 (干渉数低減操作 2) である．図 4.8 に干渉数低減操作 2 の適用例を挙げる．

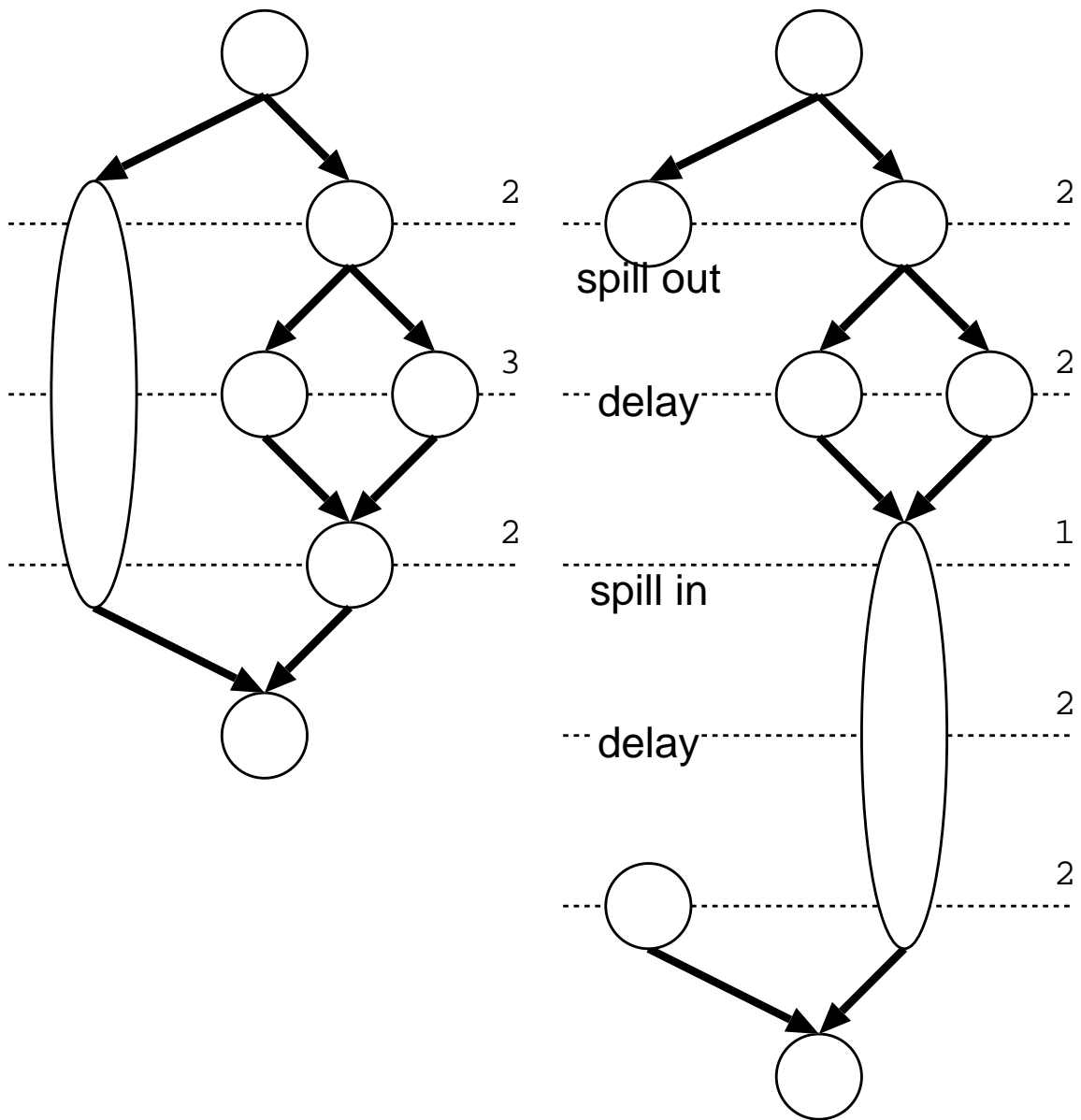


図 4.8: 干渉数低減操作 2 の例

このような例のように、長い生存区間を持っていて、かつ、しばらく参照されない仮想レジスタが存在する場合などに、有効な方法であるといえる。

これらの操作によって、プログラムの実行時系列の要素数が変わる可能性があるため、すべてのノードにおいて自由度の再計算が必要である。また、同時には起こりえないガードを持つ仮想レジスタ同士は、論理排他性を利用することで、同じ物理レジスタに割り付けることができる。例えば、(cc1) と (!cc1) というガードを持つ仮想レジスタ同士や、複数のガードが組み合わさった (cc1&cc2) と (cc1&!cc2) というガードを持つ仮想レジスタ同士などが、これに当てはまる。

最大干渉数低減操作のアルゴリズムとしては、次のようになる。レジスタ生存グラフの深さ 1 から始めて、グラフ上の時刻をサイクルとして次のことを繰り返す。

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

図 4.6 のレジスタ生存グラフに対して、レジスタ数を 4 として最大干渉数低減操作を施したものを、図 4.9 に示す。

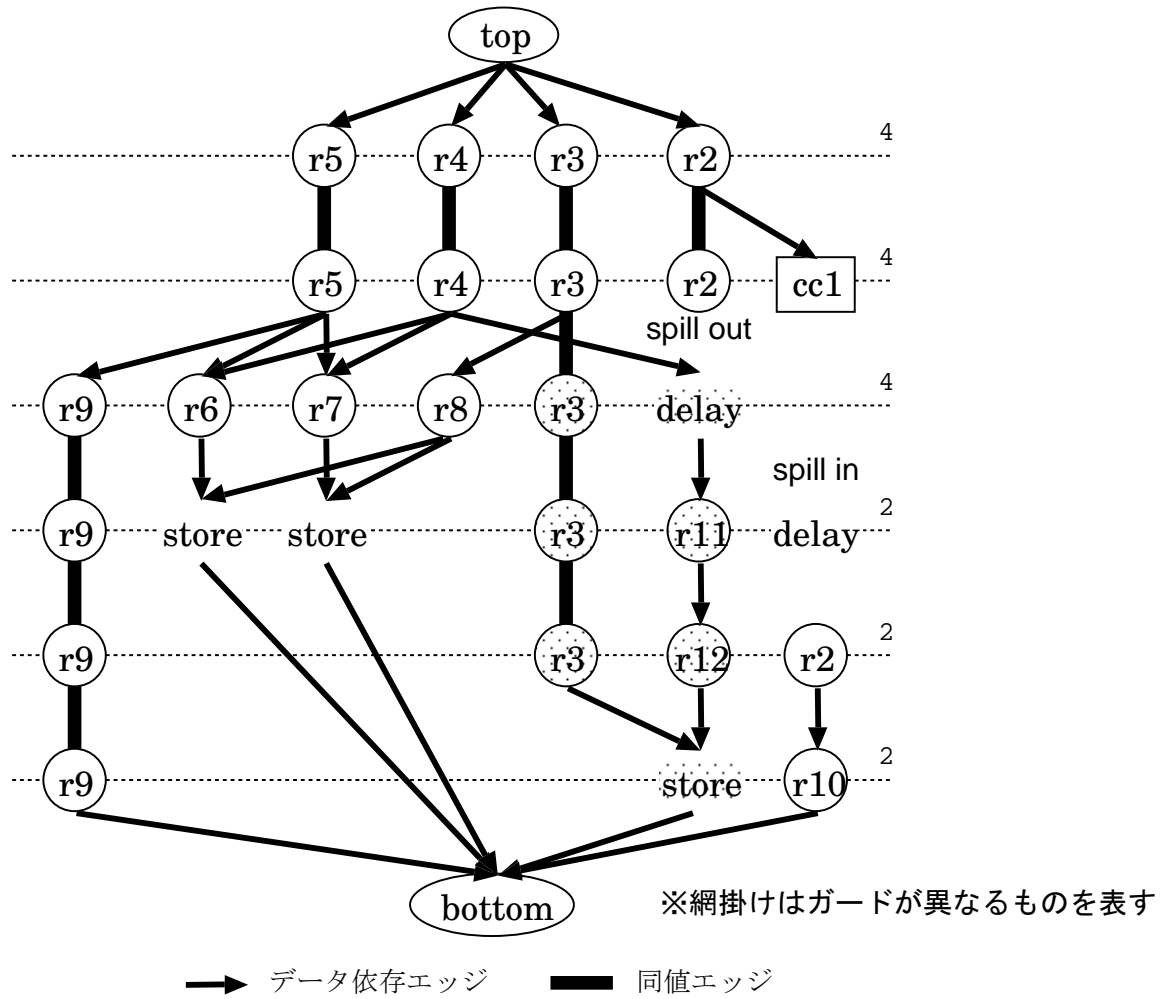


図 4.9: 最大干渉数低減操作を施したレジスタ生存グラフ

図 4.9 の網掛けになっている仮想レジスタには論理排他性があり，網掛けになっていない仮想レジスタと同じレジスタを共有することが可能になっている．これを利用すると，すべてのサイクルにおいて干渉数が 4 以下になっている．

4.3.4 プレスケジューリング

最大干渉数低減操作は、機能ユニットが無限に存在すると仮定した場合に、物理レジスタが不足しないという保証をするものということである。しかし、実際には機能ユニット数を超える演算の組み合わせを、同時実行することはできない。そのため、機能ユニット数やデータ依存エッジによって作られるグラフの構造によっては、機能ユニットの割付け時に物理レジスタが足りなくなることがある。このような場合、コードスケジューラはnop命令やスピルコードを挿入するために、最大干渉数低減操作を施した時に想定していた性能を発揮できなくなる。

プレスケジューリングの方法は以下の通りである。

1. 作業用リストに、現在のサイクルで発行可能な命令をすべて入れる。
2. 作業用リスト内の命令を、自由度の低い順に並べ替える。
3. 作業用リストから、与えられた機能ユニット数で考えられる演算の組み合わせの中で、自由度の合計が最低になるものをすべて作る。
4. (3) で作成したすべての組み合わせに対して、命令実行後の干渉数を計算する。
5. 干渉数が物理レジスタ数を超えているものがない場合は、以下の操作を行う。
 - (a) 組み合わせの中に入っているすべての命令を、作業用リストから除く。
 - (b) 作業用リスト中の命令の自由度を1減らし、発行可能な命令がなくなるまで(1)へ戻る。
6. すべての組み合わせにおいて、干渉数が物理レジスタ数を超えている場合、以下の操作を順に試す。
 - (a) 自由度の合計が最低にならないの演算の組み合わせを作り、(4)に戻る。

- (b) nop 命令や，利用されるまでの時間が最も長い値のスピルアウト命令との組み合わせを作り，(4) に戻る．スピルアウト命令挿入の場合には，2 サイクル後にスピルイン命令を挿入する．
 - (c) 1 つも命令が発行できない場合，スピルアウト命令・スピルイン命令を挿入し，レジスタ生存グラフの再構築，命令の自由度の再計算をして，(1) に戻る．
7. 干渉数が物理レジスタ数を超えているものが 1 つ以上ある場合，以下の操作を行う．
- (a) 命令実行後の干渉数が物理レジスタ数を超えている組み合わせが，すべて起きないようにレジスタ生存グラフを変形する．
 - (b) 変形によって，生存区間が変化しなかった仮想レジスタを生成する命令を，作業用リストから除く．
 - (c) レジスタ生存グラフの整合性がなくなるため，レジスタ生存グラフの再構築をする．
 - (d) レジスタ生存グラフを再構築したことによって，命令の自由度も変わるので，自由度を再計算して，(1) へ戻る．

図 4.9 のレジスタ生存グラフに対して，機能ユニット数を 2 としてプレスケジューリングを行ったものを図 4.10 に示す．

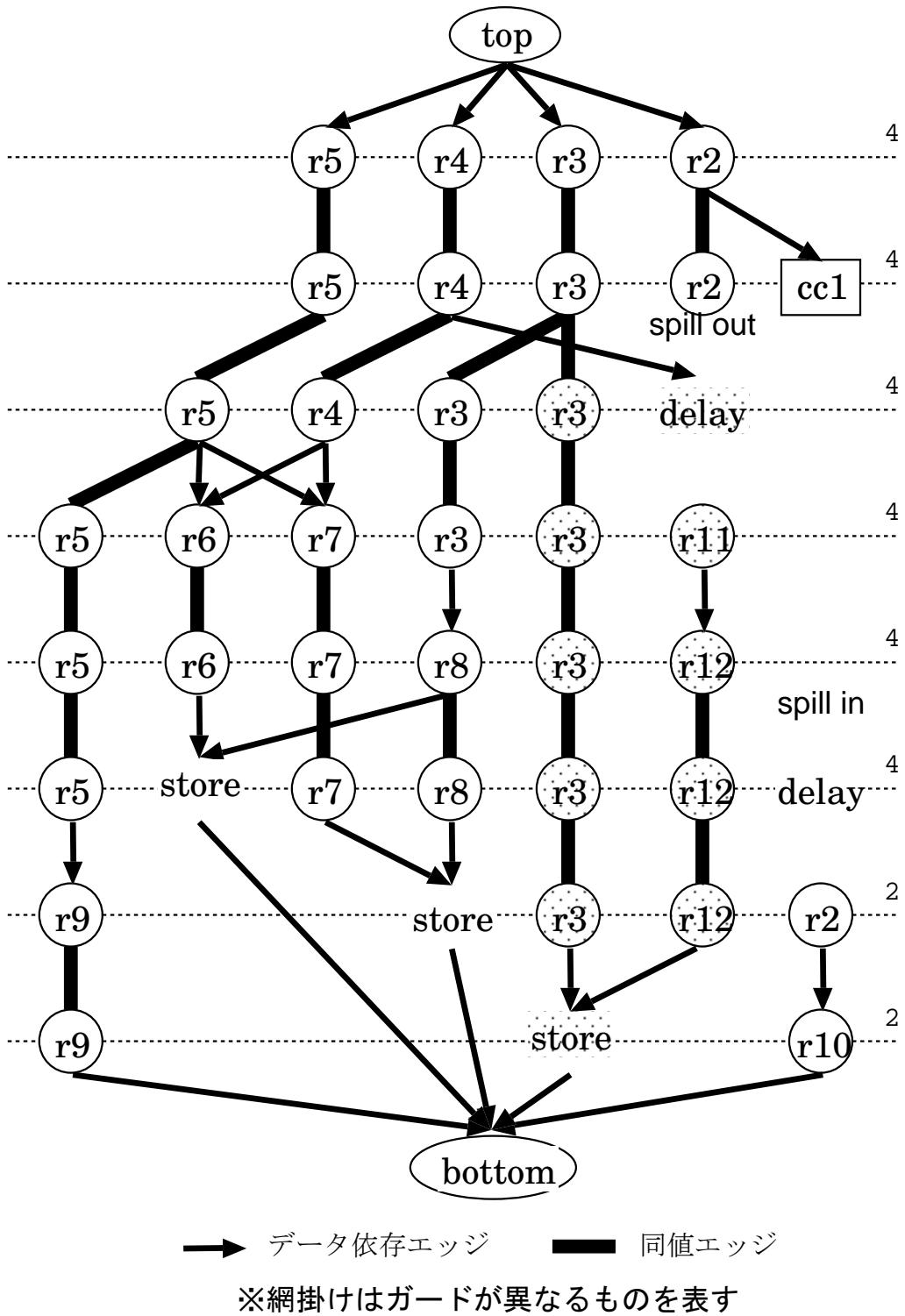


図 4.10: プレスケジューリングを施したレジスタ生存グラフ

この変形により，機能ユニット数が2ならばレジスタ数が4で不足しないことが保証される．

また，ここまでレジスタ生存グラフに対して施した変更を，GPDGにも反映させる．図4.5のGPDGにプレスケジューリングまでの変更を加えたGPDGを，図4.11に示す．

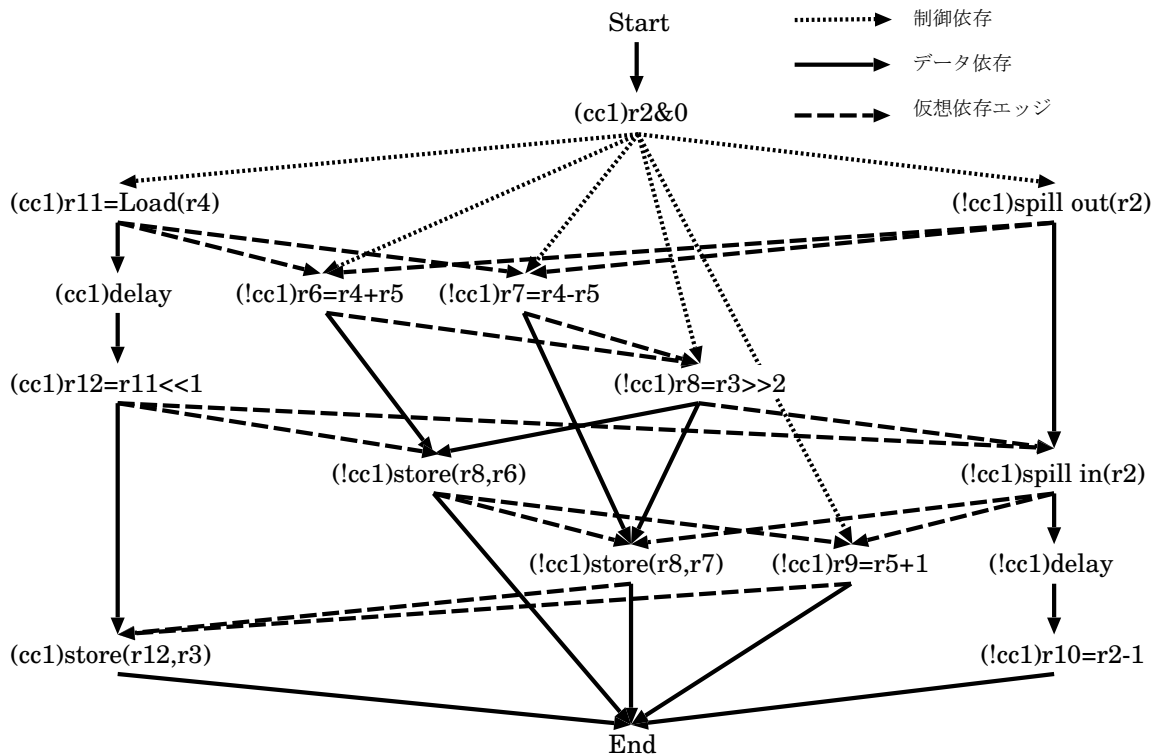


図 4.11: プレスケジューリングまでの変更を加えた GPDG

レジスタ割付けアルゴリズム

変形後の GPDG 上でノードとして表されている各命令に対して，命令の並べ替えと同時に仮想レジスタを実レジスタに割付ける．コードスケジューラは，命令レベルでの並列性を抽出し，GPDG のクリティカルパスを短縮するために，スピルコードの生成を試みる．具体的な割付けは，以下に示すようなスピルコードの生成とレジスタの割付けアルゴ

リズムを追加した，クリティカルパスリストスケジューリングによって行う．

1. カレントマシンサイクルを1とする．
2. 現在の深さを1とし，作業用リストにGPDG上の深さ1のノードをすべて入れる．
3. スタートノードにおいて，すでに生存している仮想レジスタを実レジスタに割付ける．
4. 作業用リストが空になるまで以下の操作を繰り返す．
 - (a) 作業用リストのノードを自由度の低い順，かつ，用いるレジスタ数の多い順にソートする．
 - (b) 作業用リストの先頭から機能ユニット数を超えないように，カレントマシンサイクルの命令スロットに割付ける．
 - (c) 割付けたノードが定義している仮想レジスタを実レジスタに割付ける．
 - (d) 命令スロットに空きがある場合，スピルコードを生成して時間の短縮を試みる．
 - i. スピルコードによって移動される命令の候補は，GPDGにおける仮想エッジの終点ノードである．これらのノードを自由度でソートする．
 - ii. 各ノードに対し，スピルコードを生成した場合のGPDGのクリティカルパス長を計算する．このクリティカルパス長が，元のGPDGのクリティカルパス長より短くなっていない場合は，スピルコードの生成を取りやめる．
 - iii. スピルコードのための命令を生成し，命令スロットに割付ける．
 - (e) 作業用リストにある各ノードの自由度を1減らす．
 - (f) 現在の深さを1増やし，作業用リストにその深さのノードを加える．
 - (g) マシンサイクルを1増やす．

4.4 実験

ここでは、本手法を用いたレジスタアロケータ及びコードスケジューラの評価を行う。ハードウェアは、IA-64 に準拠したプロセッサとし、乗除算を除く整数演算命令と Store 命令には 1 クロック、Load 命令には 2 クロックかかるものとした。また、レジスタ割付けに対する定常的な実験をするため、キャッシュミスは起こらないものとした。このような条件下で、レジスタ数が 8, 32, 機能ユニットの数が 2, 4 の各場合について実験を行った。評価プログラムとして、SPECint2000 に含まれるプログラムのソースコードから、従来手法でコード生成時にレジスタが足りなくなる可能性があり、本手法を適用できる最内ループを抽出して用いた。使用したプログラムは、gcc bc-optab.c, voltex tree0.c, twolf clean.c, twolf findest.c となっている。

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

従来手法として、文献 [5] で挙げられている手法でクリティカルパス長を求め、我々の手法を適用したときの性能と比較した。ただし、従来手法ではクリティカルパス長が割付け方法によって変化するため、確率変数をクリティカルパス長とし、その割付け方法が起こる確率を求めて、期待値を評価値としている。レジスタ数が 8 個の場合での実行サイクル数を表 4.1 に、レジスタ数が 32 個の場合での実行サイクル数を表 4.2 に示す。ただし、レジスタ数が 32 個の場合は、32 個すべてのレジスタを使いきることがなかったため、表 4.3 に使用した最大レジスタ数を示す。また、従来手法の並列度 2 の場合を 1 として、それぞれ比率にしたグラフを図 4.12, 図 4.13, 図 4.14 に示す。つまり、棒の短い方が性能がより良いことを示している。

表 4.1: レジスタ数 8 の場合の実行サイクル数

並列度	従来手法		本手法	
	2	4	2	4
gcc bc-optab.c	32	21	30	21
voltex tree0.c	16	7	11	7
twolf clean.c	27	17	26	17
twolf findest.c	9	8	8	8

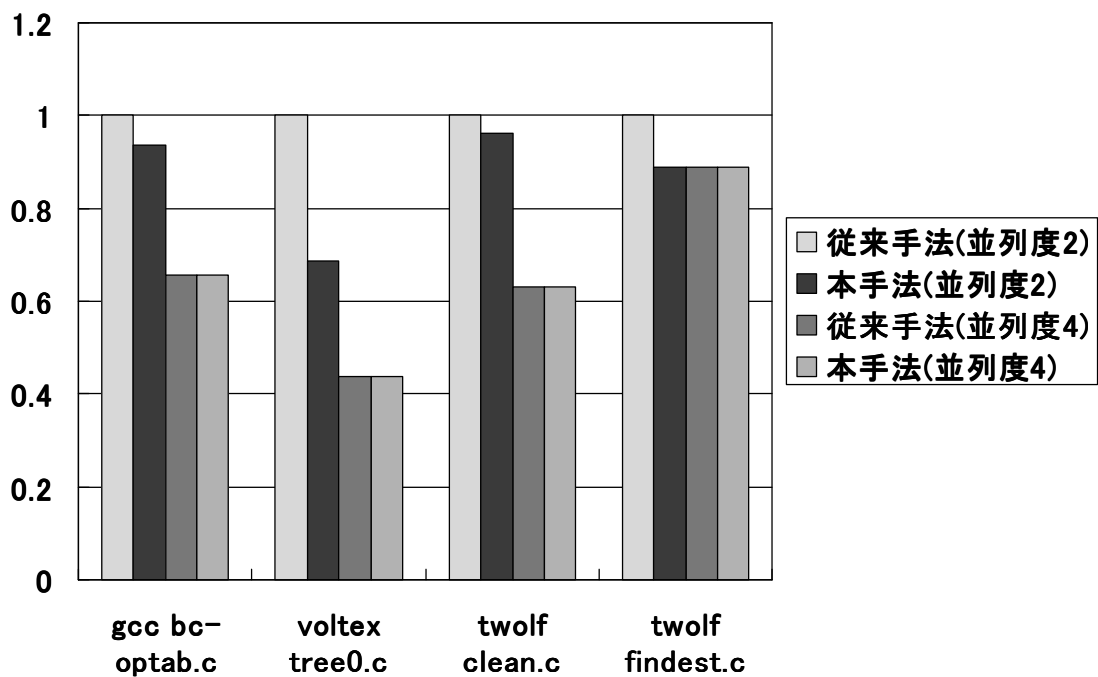


図 4.12: レジスタ数 8 の場合の実行サイクル数比率

表 4.2: レジスタ数 32 の場合の実行サイクル数

並列度	従来手法		本手法	
	2	4	2	4
gcc bc-optab.c	28	19	28	19
voltex tree0.c	11	6	11	6
twolf clean.c	26	17	26	17
twolf findest.c	8	8	8	8

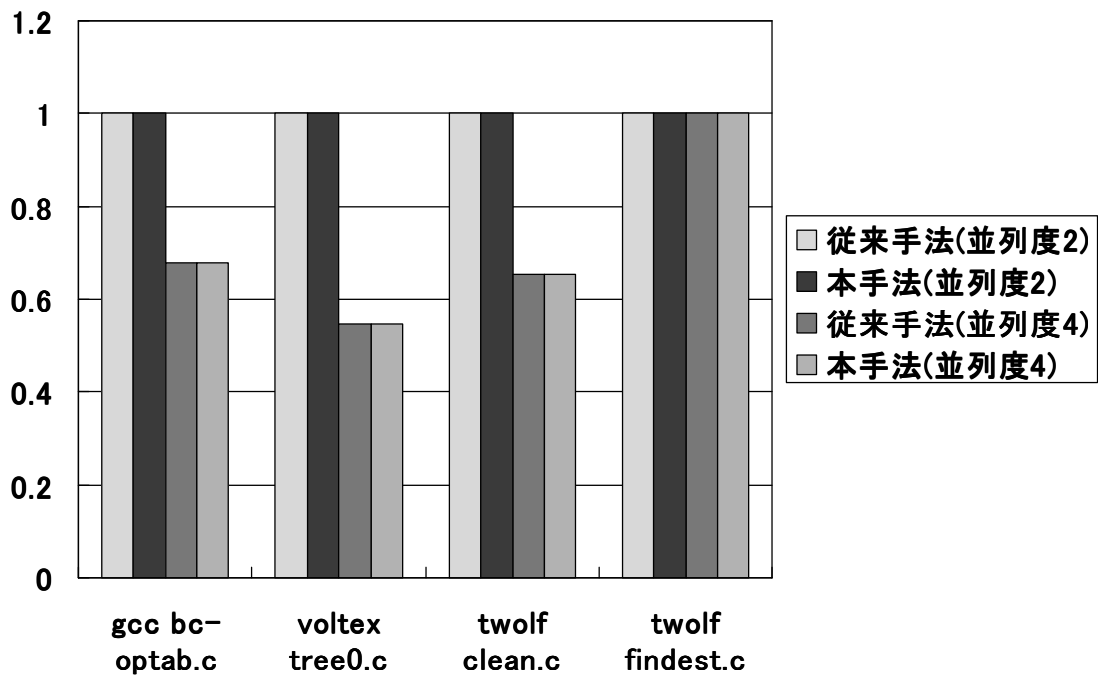


図 4.13: レジスタ数 32 の場合の実行サイクル数比率

表 4.3: レジスタ数 32 の場合の使用レジスタ数

並列度	従来手法		本手法	
	2	4	2	4
gcc bc-optab.c	14	12	12	11
voltex tree0.c	10	9	9	9
twolf clean.c	11	11	11	11
twolf findest.c	9	9	8	8

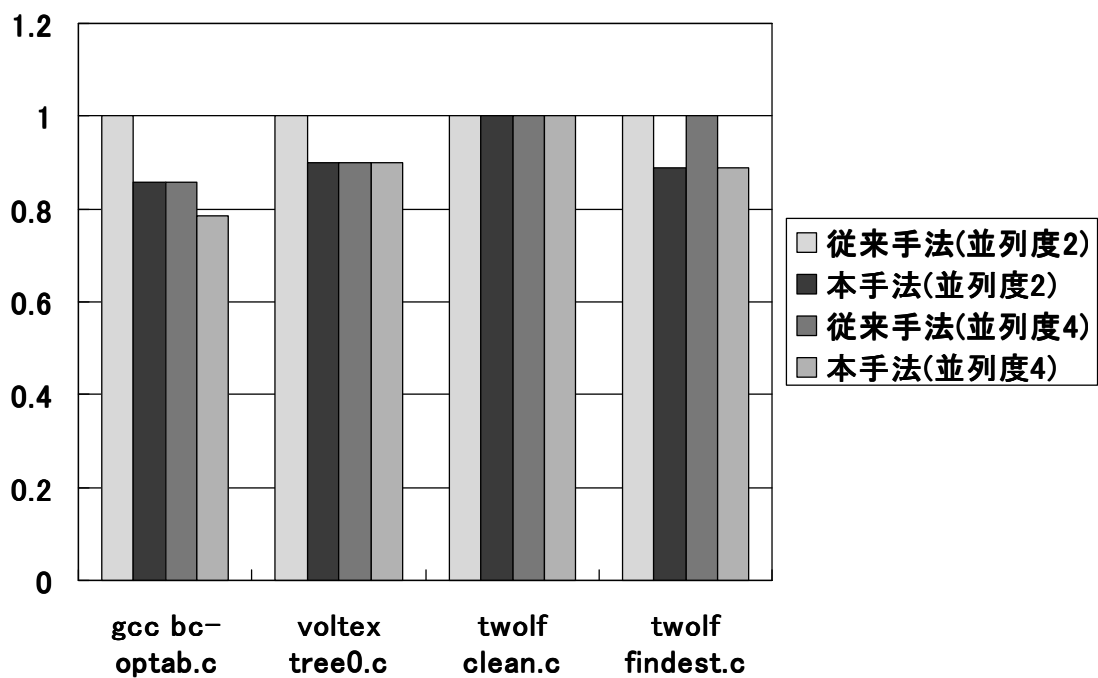


図 4.14: レジスタ数 32 の場合の使用レジスタ数比率

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

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

最後に，使用できるレジスタ数が多い場合には，並列度に関係なく，レジスタが不足する状況が起こりえなかったため，実行サイクル数は同じとなっている．しかし，最内ループ内で使用するレジスタ数を比較してみると，本手法の方が少なくなっている．これは，レジスタ生存グラフ上で，機能ユニットの数の考慮がされていないために起こるためと考えられる．これにより，本手法はループアンローリングや Lazy Code Motion[12] といった，使用レジスタ数を増やす最適化と相性が良いと考えられる．例えば，2,3 回ループアンローリングを行えば，レジスタ数 32 でもレジスタが足りなくなる．この場合には，DOALL 型ループであっても DOACROSS 型ループであっても，レジスタ数 8 の場合と同じように実行サイクル数に差が出ると思われる．

また，本手法の計算量と性能のトレードオフについてだが，本手法の計算オーダーは，同時演算可能な演算数の組み合わせの数に等しく，指数関数並と非常に大きい．しかし，探索範囲は非常に狭く，また実験により，並列度が低い場合には実行サイクル数の削減，並列度が高い場合には使用される物理レジスタ数の削減と，実行するだけ価値がある性能差が現れていると思われる．

4.5 関連研究

4.5.1 並列化干渉グラフを用いたレジスタ彩色法

命令レベル並列プロセッサ向けレジスタ割付け手法としては，並列化干渉グラフを用いたレジスタ彩色法 [6][7] が挙げられる．この手法は，レジスタ割付け手法としては最も有名なグラフ彩色法に対して，並列化の考慮を加えて拡張した手法である．並列化の考慮として，並列性を保持するエッジを干渉グラフに対して加え，彩色問題を解くことでレジ

スタ割付けを行っている。この手法の長所としては、干渉グラフを生成する部分を拡張するだけで、グラフ彩色法の構造化されたレジスタ割付け手法をそのまま利用できる点である。

しかし、プログラムの中間表現が中間コードの命令の並び順に依存しているため、並列化干渉グラフ上でノード間にエッジの張り方が一通りにならない問題があった。また、並列性を考慮するエッジも干渉エッジと同じ扱いをするため、干渉グラフが完全グラフに近くなる問題もあった。この理由は、この手法が並列実行に対して安全な見積もりを行っているためである。ゆえに、現実には干渉しないノード間にもエッジが張られたままになることが多く、無駄なスピルコードが挿入されてしまうという問題があった。さらに、レジスタ割付けとコードスケジューリングの相互干渉については、まったく考慮されていなかった。

ここで、並列化干渉グラフがどれほど完全グラフに近くなるか、例を示す。まず、図 4.15 のサンプルコードに対する干渉グラフを図 4.16 に示す。

```
if(r2 != 0){
    r6 = r3<<2
    store(r6,r3)
}else{
    r7 = r4>>2
    r6 = r4+4
    store(r7,r6)
    r9 = Load(r5)
    r10 = r9+1
    store(r10,r5)
    r11 = r2+1
}
```

図 4.15: 並列化干渉グラフがほぼ完全グラフになるサンプルコード

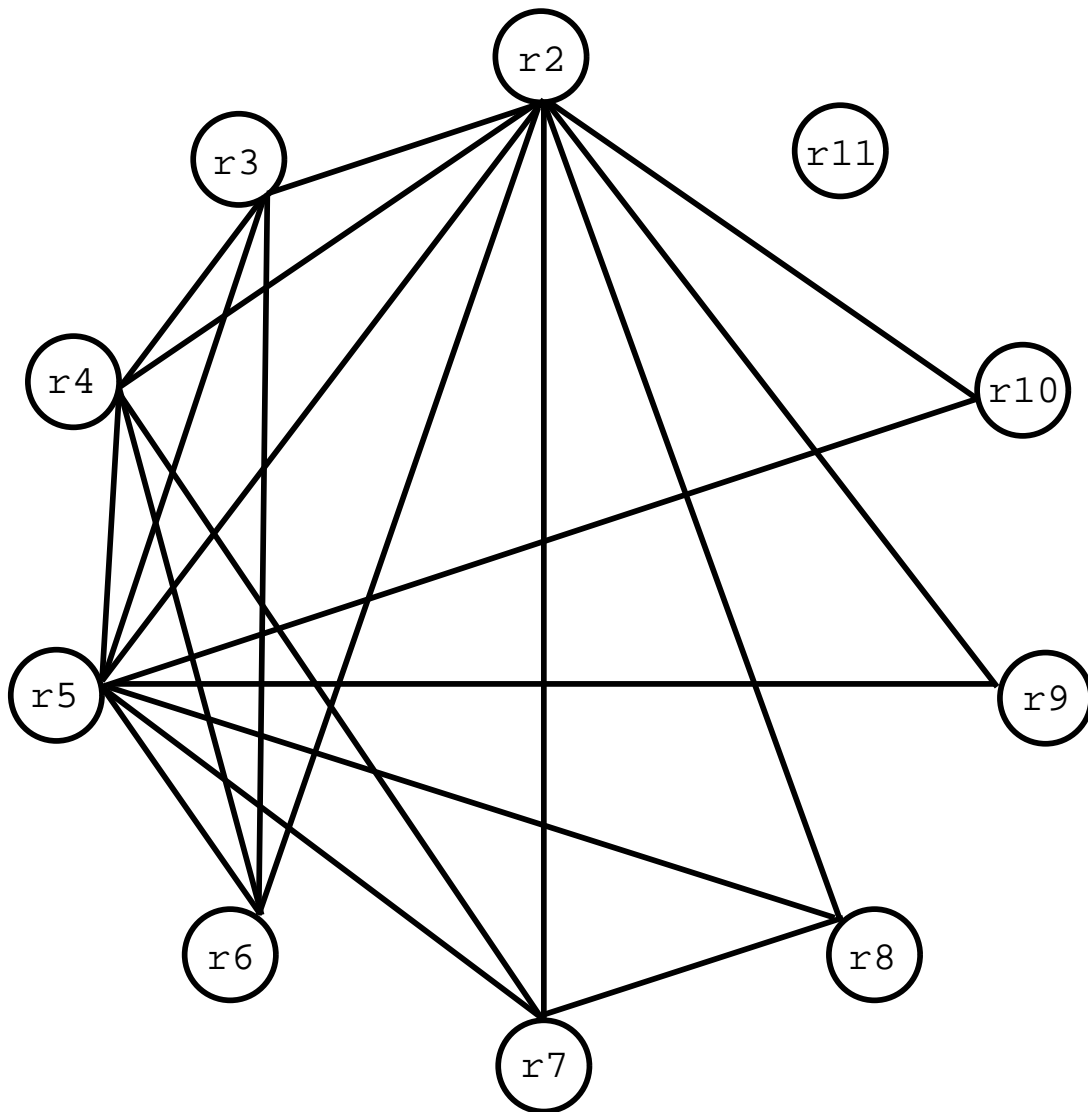


図 4.16: 干渉グラフの例

各仮想レジスタをノードで表し，生存区間が重なっている仮想レジスタを表すノード間には，干渉エッジと呼ばれる無向エッジで結ばれている．干渉グラフ，つまり，並列性の考慮を加えない時点では，まだ完全グラフには遠いことがわかる．

次に，図 4.15 のサンプルコードに対する並列化干渉グラフを図 4.17 に示す．

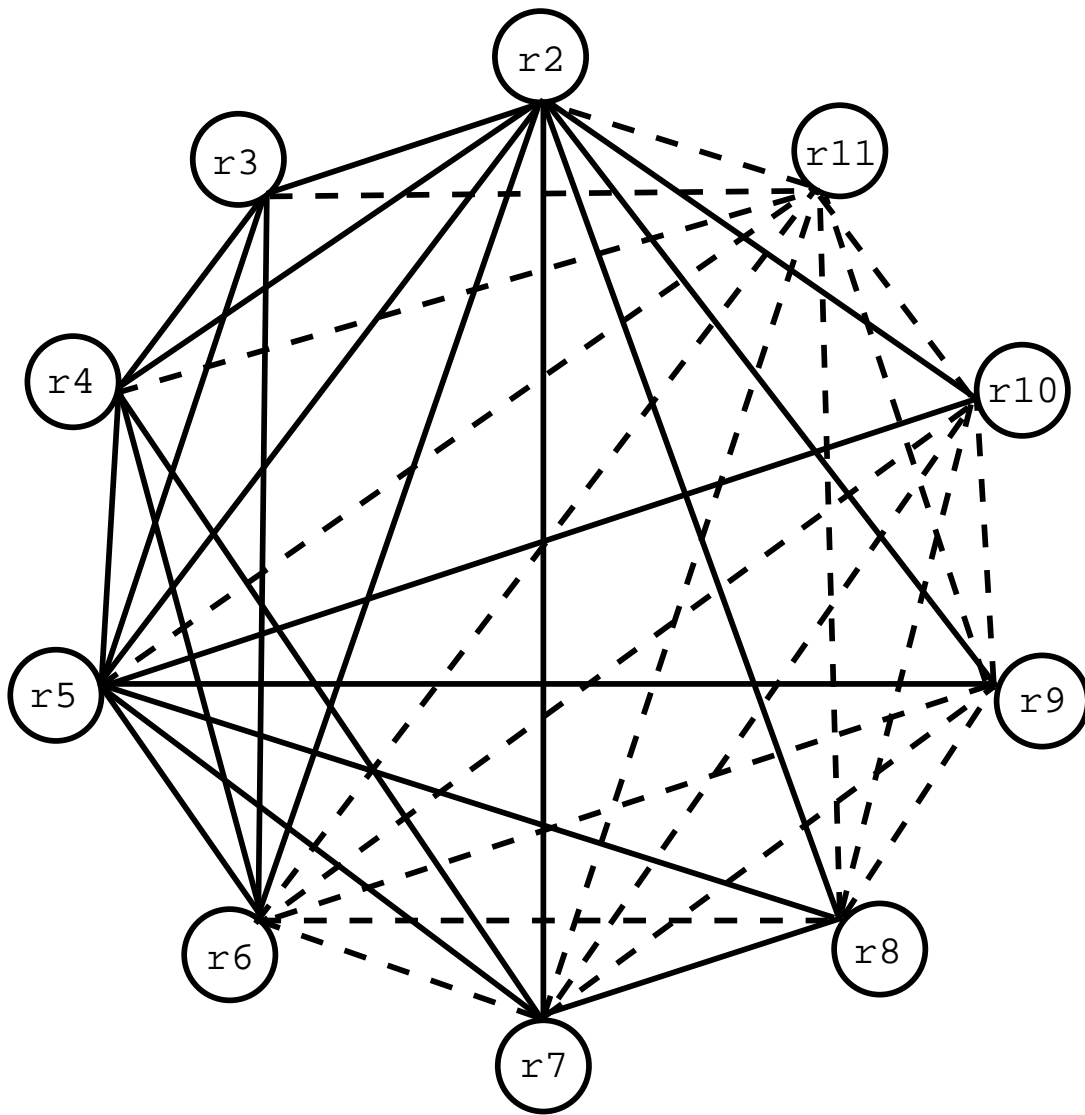


図 4.17: 並列化干渉グラフの例

ノードと実線については、干渉グラフと同じものを指している。新たに増えた点線のエッジが、並列性を考慮を表しているこのように、並列性考慮するエッジが追加されたことにより、ほぼ完全グラフとなっており、彩色が困難になっているのがわかる。

4.5.2 Integrated Code-Scheduling

一方，レジスタ割付けとコードスケジューリングの相互干渉を緩和した手法として，Integrated Code-Scheduling[8] [11] が挙げられる．これらの手法は，大域的なコードスケジューリングを行ってからレジスタ割付けを行い，最後に局所的なコードスケジューリング

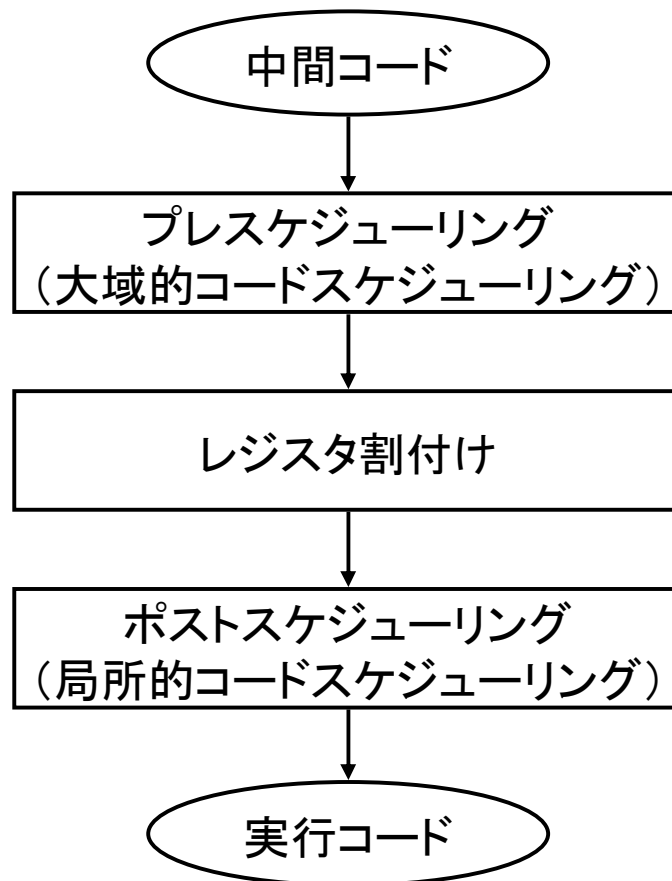


図 4.18: Integrated Code-Scheduling の流れ

特に手法 [8] の後続研究である手法 [9] では、閉路を持たない有向グラフ (DAG: Directed acyclic graph) 1 つで物理レジスタの使用状況と機能ユニットの使用状況を表し、この DAG の物理レジスタ・機能ユニットの空きに、部分 DAG を合成して埋めることで、オンザフライな機能ユニット割付けを可能としている。

しかし、手法 [8] では、レジスタ割付けの結果が中間コードの並び順に依存する問題は、解決されていない。さらに手法 [9] にも言えるが、ILP を低い状態から高めていく手法であるため、常に高い ILP を抽出できるとは限らない。また、手法 [11] では、動的コンパイラを対象としているため、時間のかかる大域的なコードスケジューリングは行っていない。

4.5.3 レジスタ生存グラフを用いた手法

レジスタ割付けの結果が中間コードの並び順に依存する問題を解決し、大域的なコードスケジューリングを行う手法として、レジスタ生存グラフを用いたレジスタ割付け手法 [5][10] が提案されている。レジスタ生存グラフは命令の依存関係から作成するため、中間コードの命令順序はレジスタ生存グラフには影響しない。そのため、このグラフは、並列化干渉グラフでは切ることが困難であった、実際には干渉しないノード間のエッジを切ることができるという特徴を持っている。

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

しかし、この制約によって保証されることは、各サイクルに存在する命令がすべて同時に実行された場合に物理レジスタが足りる、つまり物理レジスタ数制約を満たすということである。実際に同時実行できる命令数は機能ユニットの数までであるため、物理レジ

スタ数制約を満たすための操作で機能ユニットの数の考慮がされていないこの手法では、コード生成時に物理レジスタが足りなくなる恐れがあった。この場合には、コードスケジューラによる何もしない命令である `nop` 命令の挿入、もしくは物理レジスタの値をメモリに退避するスピルコードの挿入が行われてしまい、想定していた性能が出ないことがあるという欠点があった。

4.6 あとがき

本章では、命令レベル並列プロセッサ向けにコードを最適化するレジスタ割付け及びコードスケジューリング技法を提案した。プレスケジューリングを用いることで、コード生成時にレジスタが不足する可能性の低減を図った。実験の結果、実行サイクルにおいて平均約 8% の性能向上が確認できた。今後は、使用レジスタ数を増やす最適化手法と組み合わせることで、どれほど性能が向上するかを検証してみたいと考えている。

第5章 コンパイル時間と性能の両立を 図ったレジスタ割付け技法

5.1 まえがき

プログラムを高速実行するためには、コンパイラによる最適化が重要である。その重要な最適化手法のひとつに、メモリへのアクセス時間を減少させるレジスタ割付けがある。このレジスタ割付けは重要な最適化であるため、今まで多くの研究がされてきたことも事実である。しかし、我々は、レジスタ数の少ないアーキテクチャ向けの、新しいレジスタ割付け手法が必要ではないかと考えている。なぜなら、現在主流なアーキテクチャは、x86 に代表されるレジスタ数の少ないアーキテクチャであり、ほとんどすべてのプログラムにおいてレジスタが不足しているからである。つまり、対象がレジスタ数の少ないアーキテクチャの場合、メモリへのアクセス時間が増加しやすい傾向がある、ということである。また、レジスタ数の多いアーキテクチャであっても、プログラムに対して並列性を高める最適化を施すとレジスタが不足しがちになり、同じようにメモリへのアクセス時間が増加すると考えられる。

Chaitin らによって提案されたグラフ彩色法 [2][3] は、レジスタ割付け手法として有名な手法であり、現在では多くのレジスタ割付け手法の基本アルゴリズムとなっている。このようなレジスタ割付け手法では、干渉グラフを作成して、そのグラフに対する彩色問題を解くことでレジスタ割付けを行う。彩色問題を解くときに使用される色数は、対象とするアーキテクチャの使用可能なレジスタ数と等しい。この色数で変形を行っていない干渉グラフを彩色できる場合には、その解は最適解の1つであることが保証される。一方、そ

のままでは干渉グラフを彩色できない場合には、レジスタ数が不足していてすべての変数をレジスタに割付けできないことを意味する。そのため、レジスタに割付ける変数を減らすために、あるヒューリスティクスにより変数を1つ選択し、その値をメモリに退避するスピルという操作を行う。この操作によって、その変数の生存区間が変更されるため、干渉グラフを再構築するために必要となる全プロセスを、グラフが彩色できるようになるまで繰り返さなければならない。また、この操作を1回でも行った場合には、干渉グラフが彩色が可能になったとしても、最適解の1つであるという保証はできない。レジスタ数の少ないアーキテクチャが対象だった場合には、彩色に使用できる色数は非常に少なく、この繰り返し回数が増える傾向がある。そのため、これらの手法ではレジスタ割付けに時間がかかり、さらに時間をかけたにもかかわらず最適解から遠い解を出力することもある。このような理由から、最適化にかかる時間が実行時間に含まれる動的コンパイラでは、レジスタ割付けに時間がかかるこれらの手法を採用できない。それだけでなく、GNUコンパイラコレクション (GCC) の x86 を対象としたコンパイラをはじめとした、時間のかかる最適化を行える静的コンパイラでさえ、生成コードの性能面からレジスタ割付けにこれらの手法を採用していない。

そこで、我々は、レジスタ数の少ないアーキテクチャでは時間のかかる処理を繰り返さないように、アルゴリズムを変更した。さらに、同時に有効である変数の範囲に着目し、長い時間のかかるメモリへのアクセス命令を、できる限り減らそうと考えた。本章では、高速なコンパイルが可能だけでなく、高速な実行ファイルの生成も可能な新しいレジスタ割付け手法について提案していく。

5.2 本手法の概要

本手法は、レジスタ数の少ないアーキテクチャが対象の場合でも、コンパイル時間と性能の両立を図ったレジスタ割付け技法を目標としている。ここで、今までの手法の悪い点を改善するためには、干渉グラフの改善、彩色方法の改善、スピル方法の改善の3つが必

要であると考えた．それぞれについて，以下で詳しく述べる．

5.2.1 干渉グラフの改善

まず，干渉グラフの改善であるが，今までの手法よりグラフで保持する情報を増やす必要があると考えた．今までの干渉グラフでは，ノードが変数をスピルする際に必要なコストの概算を保持し，ノード間を接続するエッジについては，あるかないかの1ビットの情報しか保持していなかった．

そこで，本手法が使用する干渉グラフでは，ノードについては，表している変数の生存区間情報も保持できるようにしている．また，ノード間の接続については，1つの無向エッジではなく2つの有向エッジを双方向に張るようにし，それぞれの有向エッジに対してスピルする際に必要なコストの概算を保持できるようにしている．この有向エッジが保持するスピルする際に必要なコストの概算を，エッジコストと呼んでいる．このようにすることで，以下で説明する彩色方法の改善とスピル方法の改善が行えるようになる．

本章では，この干渉グラフのことをエッジコスト付干渉グラフ呼んでいる．

5.2.2 彩色方法の改善

次に，彩色方法の改善であるが，彩色に使用できる色数を制限する必要があるかどうか考えた．今までの彩色方法では，彩色に使用できる色数は，対象アーキテクチャの使用可能なレジスタ数以下であった．また，彩色できない場合には，必ずヒューリスティックスによって干渉グラフを変形し，彩色できるようにしていた．干渉グラフの最適な変形方法を探索する問題はNP完全問題であり，現実的な時間で解を求めるためには，何らかのヒューリスティックスが必要であるが，このヒューリスティックスが多くの場合で性能低下の原因となっている．

そこで，本手法では彩色段階でヒューリスティックスを用いることをやめて，干渉グラフが彩色できない場合には，色数を1つ増やすようにしている．このようにすることで，

彩色が必ず1回で終わることが保証され、高速化につながる。

本章では、この彩色方法を過彩色と呼んでいる。

5.2.3 スピル方法の改善

最後に、スピル方法の改善であるが、スピル操作を行うたびに生存区間解析や干渉グラフ再構築を繰り返す必要があるかどうか考えた。特に、グラフ彩色法では、ここでの繰り返しがコンパイル速度の遅い主な原因になっている。計算量で考えると、命令列長を m 、ノード数を n として、生存区間解析や干渉グラフ作成が $O(m \times n)$ 、彩色が $O(n)$ である。しかし、最悪時はすべてのノードに対してスピル操作を行う時であるから、 n 回生存区間解析や干渉グラフ作成を繰り返すことになるので $O(m \times n^2)$ となる。

そこで、本手法が使用するスピル方法は、過彩色で増やしてしまった色数を、強引にスピルコードを出しながら干渉があるノード同士を併合させることで減らすという方法で行っている。また、繰り返し回数についても、過彩色によって増やされた色数と同じ回数だけ、つまり、定数回のスピル操作を行えばよいことがわかる。さらに、スピル対象についても、より解析的に決定している。従来では、干渉グラフのノードにコストを付け、そのコストの低いものからスピルしていた。この方法は、使用頻度は少ないが生存区間が長いような変数がスピル対象に選ばれにくく、よいスピル対象の選び方ではなかった。そこで、本手法では、エッジコストの低いエッジからスピル対象を選んでいる。例えば、 a から b へのエッジのエッジコストが最小であるとき、スピル対象は b となる。このようにすることで、結果としてスピルコードの少ない実行コードの生成が図れる。

本章では、このスピル操作をスピル付併合と呼んでいる。

5.3 本手法の流れ

ここでは，サンプルプログラムから，コード生成が行われるまでの本手法の流れを示す．本手法の全体の流れを，図 5.1 に示す．

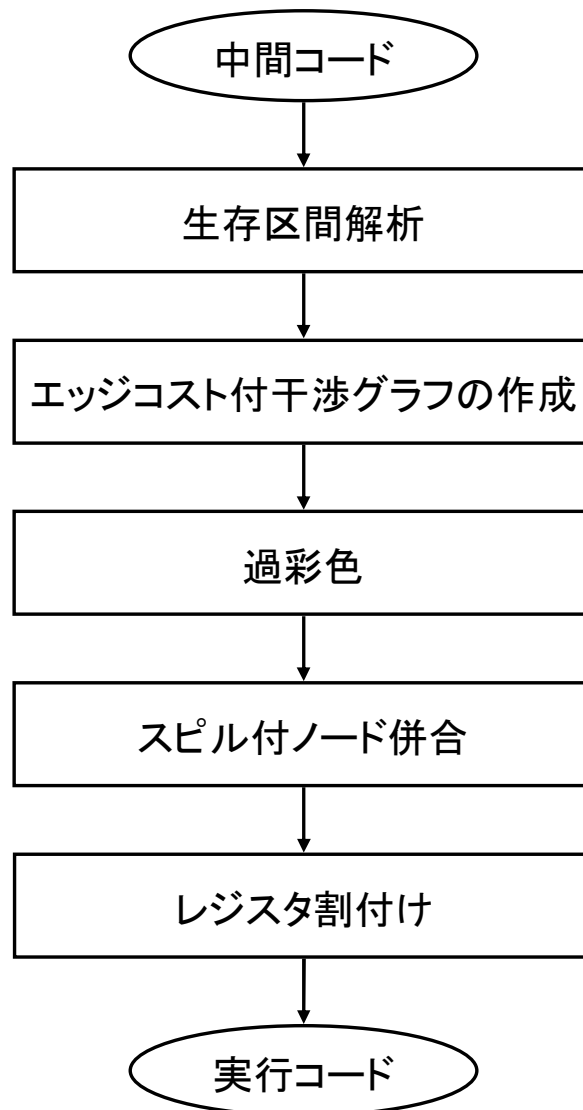


図 5.1: 本手法の流れ

本手法の処理は、このようなステップに分けることができる。まず中間コードから、生存区間解析を行い、エッジコスト付干渉グラフを作成する。このエッジコスト付干渉グラフに対して、過彩色を行う。過彩色の結果、使用された色数が物理レジスタ数以下ならば、そのままレジスタ割付けを行う。使用された色数が物理レジスタ数を越えた場合には、スピル付併合を行う。

それぞれについて、以下で詳しく説明する。

5.3.1 生存区間解析

まず、中間コードの各命令から、各変数が生成されてから最後に使用されるまでの、有効な値を保持している区間を解析する。プログラムの中間コードから CFG の構築が完了している場合、この解析は以下の式を用いて計算することができる。このとき CFG のノードは、基本ブロック単位でも命令単位でもよい。しかし、本手法では命令単位での生存区間情報が必要なため、基本ブロック単位で CFG を構築している場合には、各基本ブロック内でも計算が必要になる。

$$in[n] = use[n] \cup (out[n] - def[n])$$

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

ただし、 n は CFG のノード、 $in[n]$ はノード n に有効な値を保持して入ってくる変数の集合、 $out[n]$ はノード n から有効な値を保持して出ていく変数の集合、 $pred[n]$ はノード n の CFG 上で1つ前に位置するノード集合、 $def[n]$ はノード n で定義される変数の集合、 $use[n]$ はノード n で使用される変数の集合である。この式を、すべての n について $in[n] = \quad$, $out[n] = \quad$ から開始し、すべての n について $in[n]$, $out[n]$ を計算して、その結果が1回前の計算結果と同じになるまで繰り返す。つまり、収束したときに得られる解が生存区間解析の結果となる。また、このようにして得られた解は最小不動点と呼ばれ、この式が収束する最も小さな解であることが証明されている。最も小さな解というのは、

上式が収束する解に冗長な要素を加えたものも上式の解であるが、冗長な要素がまったくない解のことである。

一般的に、CFG の最初から始めて前向き (in, out の順) に計算を繰り返すより、CFG の最後から始めて後ろ向き (out, in の順) に計算を繰り返す方が、早く解が収束することが知られている。この理由としては、in が同じノード番号の out を参照している点と、out が CFG 上で 1 つ先の in を参照している点が挙げられる。また、後ろ向きに計算した場合、バックエッジがなければ生存区間解析の 1 回目と 2 回目の結果は同じになる。バックエッジがある場合でも、あるバックエッジの作るループの内側から別のバックエッジが外側へ飛び出すという図 5.2 の太線のような構造がない限りは、生存区間解析の 2 回目と 3 回目の結果は同じになる。CFG の構造から、生存区間解析が何回目で終わらせてよいかということについては、文献 [13][14] に記されている。

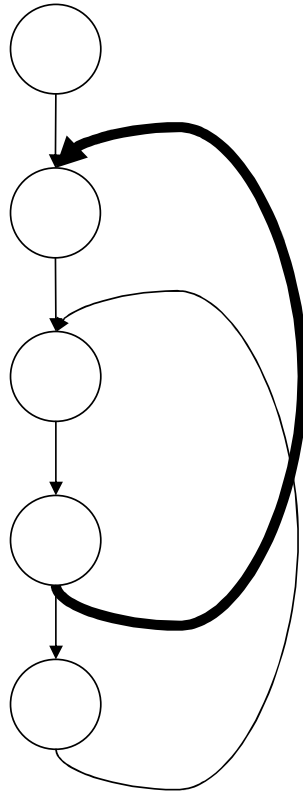


図 5.2: 生存区間解析が苦手とする構造

ここで、生存区間解析の例を示す。図 5.3 のサンプルコードに対して、CFG の最後から後ろ向きに生存区間解析を行った様子を表 5.1 に示す。表の中で、前回の `in`、`out` の値から変更になった要素を太字で示した。また、9 番目の命令で定義される変数 `D` については、後続命令で使われるものと仮定した。


```

1:      A=E+E
2:      A=A*B
3:      C=A/E
4:      F=1
5:      D=C+A
6: LOOP: D=D+F
7:      F=F+1
8:      IF F<E GOTO LOOP
9:      D=E/D

```

図 5.3: サンプルコード

表 5.1: 生存区間解析の経過

ノード 番号	def	use	0回目		1回目		2回目		3回目	
			in	out	in	out	in	out	in	out
1	A	E			BE	ABE	BE	ABE	BE	ABE
2	A	AB			ABE	AE	ABE	AE	ABE	AE
3	C	AE			AE	ACE	AE	ACE	AE	ACE
4	F				ACE	ACEF	ACE	ACEF	ACE	ACEF
5	D	AC			ACEF	DEF	ACEF	DEF	ACEF	DEF
6	D	DF			DEF	DEF	DEF	DEF	DEF	DEF
7	F	F			DEF	DEF	DEF	DEF	DEF	DEF
8		EF			DEF	DE	DEF	DEF	DEF	DEF
9	D	DE			DE	D	DE	D	DE	D

このように2回目と3回目の計算結果が完全に一致したので、これが生存区間解析の結

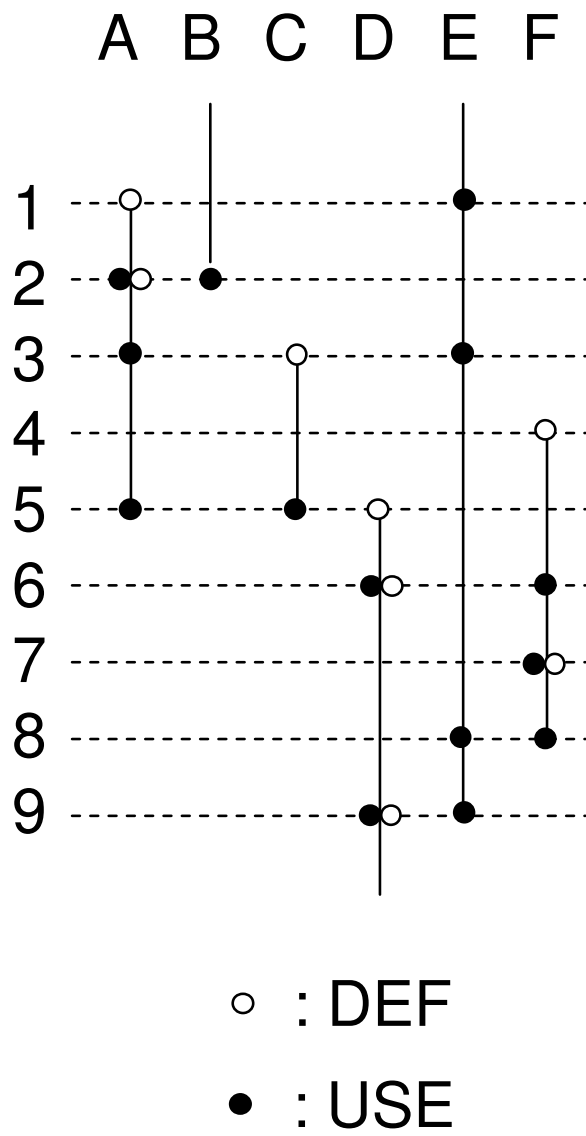


図 5.4: 生存区間解析結果

5.3.2 エッジコスト付干渉グラフの作成

ここでは、エッジコスト付干渉グラフを作成する。従来の干渉グラフと同じく、変数をノードとし、変数の生存区間が重なっている場合には、ノードをエッジで結んでいく。この干渉グラフから、エッジを双方向の有向エッジに、各エッジに変数同士の干渉度を保持させるようにすることで、エッジコスト付干渉グラフとなる。

エッジコストが表すものは、2つの変数を同じレジスタに配置する場合に、必要と見積もられる CPU サイクル数である。このため、まったく干渉がない場合には干渉度が0となり、その場合にはノード間にエッジは張られない。エッジコストが双方向である理由は、2つの変数のうち、どちらの変数を優先的にレジスタに割付けるかによって、干渉度が変わってくるためである。また、ループ内で使用される変数などは、ループを何回繰り返すかわからないことが多く、正確な CPU サイクルの見積もりは困難であるが、スパルされにくいように、コストが高めに設定される。

ここで、ABI (Application Binary Interface) などによって、特殊レジスタに割付ける必要がある変数については、特別な考慮が必要である。これに該当する変数とは、ループのカウンタや、関数の引数などに使用される変数である。このような変数を表すノードは、プリファレンス情報などを用いて、あらかじめ彩色される色が限定されるからである。特別な考慮が必要な変数を表すノード間に、エッジが張られていない場合は問題にならない。しかし、エッジが張られていた場合には、必要に応じて変数の生存区間分割を行い、ノード間にエッジが張られないようにグラフを変形する必要がある。

一方、このエッジコスト付干渉グラフ作成時の計算量については、従来の干渉グラフと同じで、命令列長を m 、変数の数を n とすれば、計算量は $O(m \times n)$ である。従来の干渉グラフを生成する場合は、縦の長さが n 、横の長さが n のビットフィールドを用意し、生存区間情報から、生存区間の重なっている変数間を表すフィールドに、フラグを立てる方法などが挙げられる。本手法の場合は、ビットフィールドの代わりに、符号なし整数の2次元配列を用い、フラグを立てる代わりに加算を用いる。よって、本手法の方が、保持す

る情報量が多い分，生成時の計算量が大きくなるように感じられるが，実は計算量は同じである．

エッジコストの計算と，エッジコスト付干渉グラフの生成を同時に行うアルゴリズムの例を示す．

1. 中間コードに現れる変数の数を n として， $n \times n$ サイズの符号なし整数の 2 次元配列を用意し，すべて 0 で初期化する．
2. 中間コードの命令を最初から順に最後まで次の操作を繰り返す．
 - (a) 命令で使用される変数のプリファレンス情報と，命令で定義される変数のプリファレンス情報を見て，同じ特殊なレジスタを使用しなければならない場合には，以下の操作を行う．
 - i. 命令で使用される変数と命令で定義される変数の両方について，現在の命令以降の区間を，特殊なレジスタに値がなければならない区間と，そのレジスタに値がなくてもよい区間に分割を行う．
 - ii. 生存区間分割に必要な命令を中間コードに挿入し，SSA 形式など分割したことによって 2 次元配列のサイズ変更が必要ならば拡張を行う．
 - (b) 2 次元整数配列上で，命令で使用される変数を表す行の，命令で定義される変数を表す列の値に， $STORE$ コスト $\times 10^{\text{ループのネスト数}}$ を加算する．
 - (c) 2 次元整数配列上で，命令で定義される変数を表す行の，命令で使用される変数を表す列の値に， $(LOAD \text{ コスト} + STORE \text{ コスト} \times 2) \times 10^{\text{ループのネスト数}}$ を加算する．

上記のアルゴリズムでは，行に表されている変数 A から列に表されている変数 B への干渉度が，2 次元配列の (A, B) 要素 (図 5.5 の灰色マス) に格納される．逆に，変数 B から変数 A への干渉度は，2 次元配列の (B, A) 要素 (図 5.5 の黒色マス) に格納される．ま

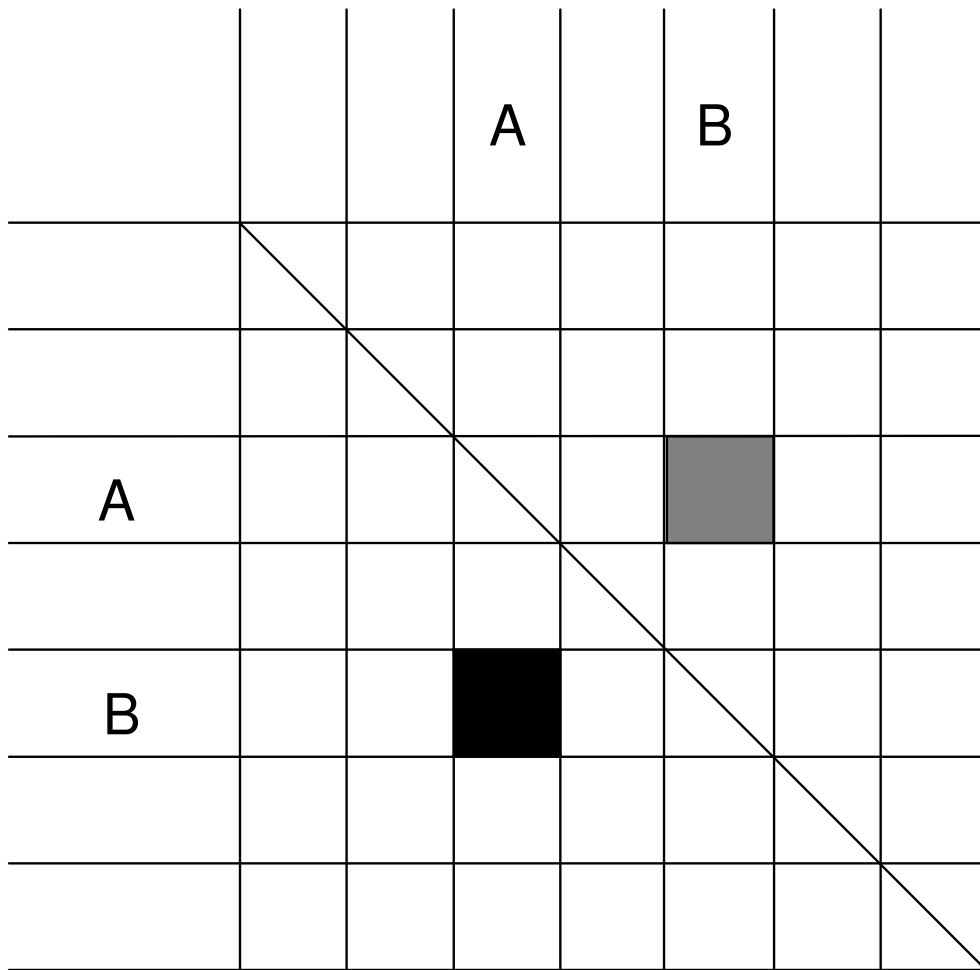


図 5.5: エッジコスト付干渉グラフの実装例

図 5.3 のサンプルコードから生成したエッジコスト付干渉グラフの例を，図 5.6 に示す．
本来は，双方向に有向エッジを張り，それぞれにエッジコストをつけるのであるが，ここ

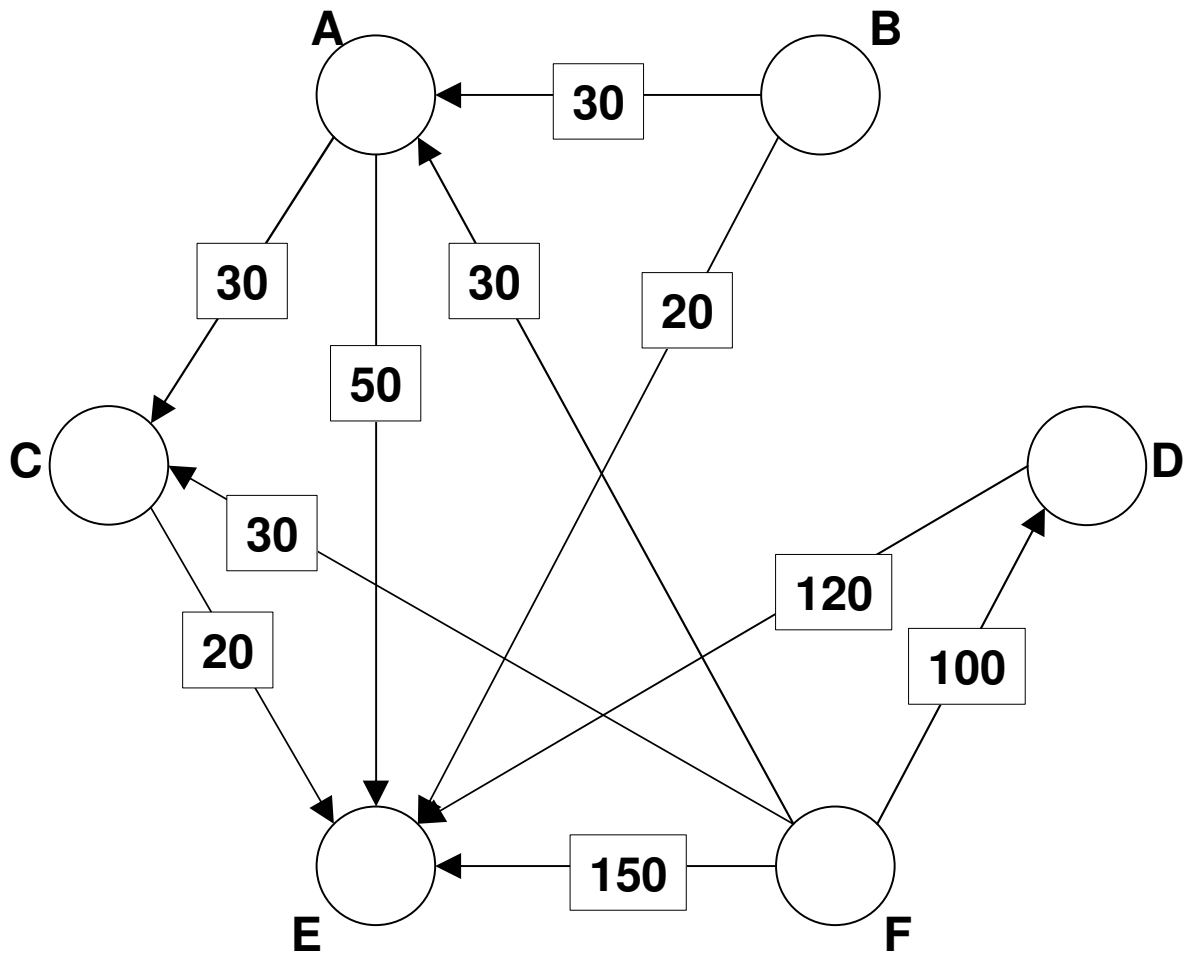


図 5.6: エッジコスト付干渉グラフの例

5.3.3 過彩色

ここでは、レジスタ数以上で、かつ、できるだけ少ない色数を用いて、エッジコスト付干渉グラフの彩色を行う。基本的には、エッジコスト付干渉グラフを何らかの方法で彩色していくことになるが、彩色できなくなった場合には、色数を増やして彩色を続ける方法

を用いる。ただし、今の色数でエッジコスト付干渉グラフを彩色できる状態で、色数を増やしてはいけない。つまり、用いた彩色方法において、できるだけ少ない色数で彩色を行わなければならない。できるだけ少ない色数でと述べたのは、与えられたグラフを彩色するために必要な最小の色数を求める問題は、NP 完全問題だからである。そのため、許容される時間でこの問題を解くためには、何らかのヒューリスティクスが必要になり、求められた解が最適解である保証はない。ただし、このヒューリスティクスが使われない場合、つまり、対象アーキテクチャにおいて、彩色に使用した色数と同じ数のレジスタがある場合は、グラフ彩色法と同様に、彩色結果が最適解であることが保証される。

ここでは、Chaitin の彩色方法を拡張して、過彩色を行うアルゴリズムを示す。まずは、エッジコスト付干渉グラフの簡略化の方法を述べる。対象アーキテクチャにおける物理レジスタの数を K とする。

1. エッジコスト付干渉グラフに、ノードから出るエッジの数が K より小さいノード N がある限り以下を繰り返す。
 - (a) N と N に出入りするエッジ群をエッジコスト付干渉グラフから削除する。
 - (b) 削除した N とエッジ群の情報をスタック S に積む。
2. エッジコスト付干渉グラフにノードが残っていない場合には、下記の過彩色アルゴリズムに遷移する。
3. 過彩色対象となるノード N を選択し、その N と N に出入りするエッジ群をエッジコスト付干渉グラフから削除する。
4. 除去した N とエッジ群の情報をスタック S に積んで、1 に戻る。

続いて、エッジコスト付干渉グラフの簡略化後に行う過彩色の方法を述べる。

1. 彩色に使用できる色数を K 色に設定する。

2. スタック S にノードがある限り以下を繰り返す。
 - (a) スタック S からノードを1つとそのノードに出入りしていたエッジ群の情報取り出して、エッジコスト付干渉グラフに加える。
 - (b) 特殊レジスタなどに割当てが必要がある変数を表しているノードだった場合は、あらかじめ決められた専用色で彩色して、2へ戻る。
 - (c) エッジで接続されているノードと異なる色がある場合には、加えたノードをその色で彩色して、2へ戻る。
 - (d) エッジで接続されているノードと異なる色がない場合には、色数をインクリメントして増やした色を加えたノードにつけ、2へ戻る。

図5.6のエッジコスト付干渉グラフに、レジスタ数を3として過彩色を行った例を、図5.7に示す。この場合では、エッジコスト付干渉グラフの彩色を3色で開始するが、彩色できないので色数が1つ増えて、4色で彩色している。

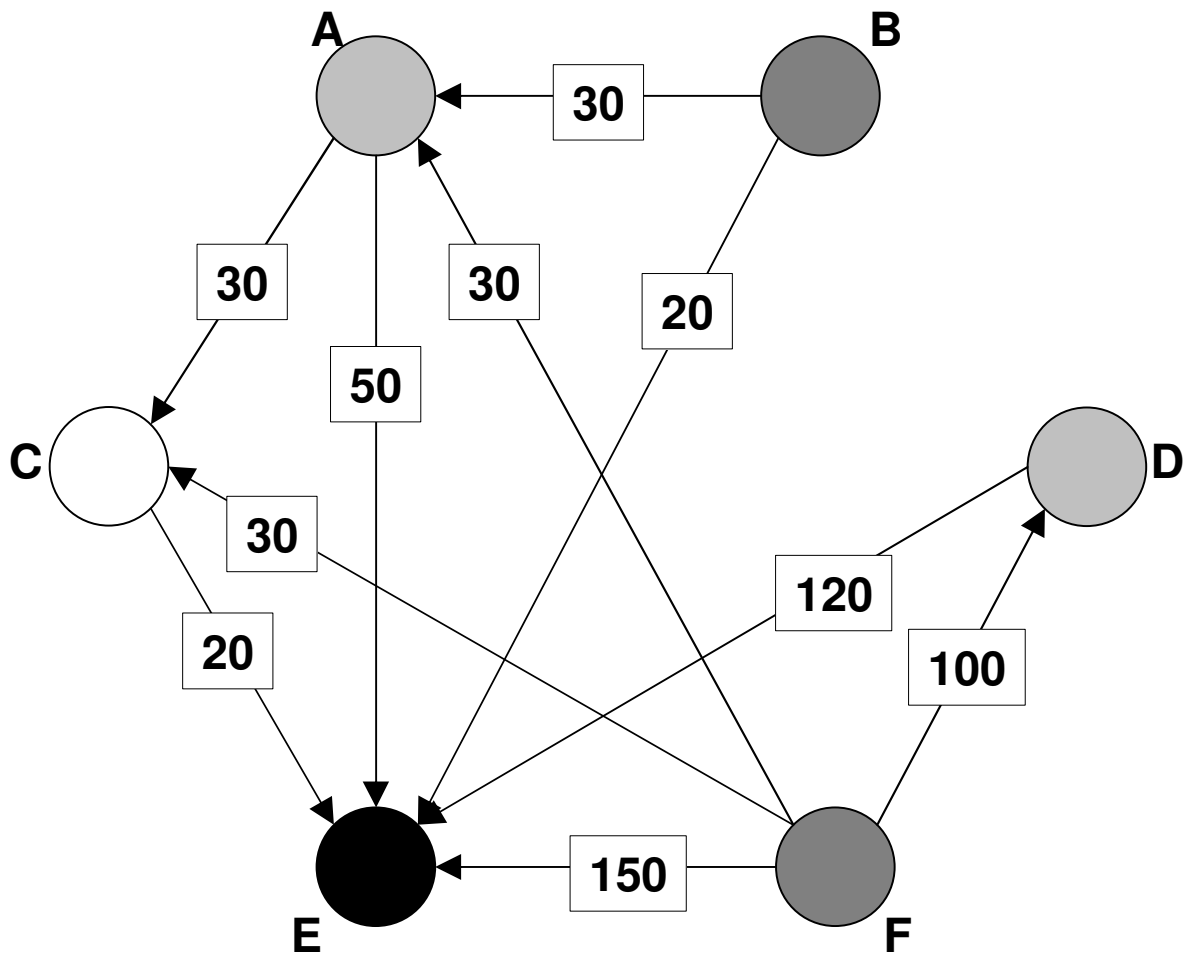


図 5.7: 過彩色の例

5.3.4 スピル付併合

ここでは、過彩色時に増やされた色数と、同じ色数だけ色を減らす作業を行う。スピルにかかるコストの小さい変数に対して、スピルを行うことによって他の変数と併合できるようにし、色数をレジスタ数まで減らす作業を行う。言い換えれば、色と物理レジスタの対応さえ取れば、レジスタ割付けが完了するところまで行う。ここでは、エッジコスト

付干渉グラフの変形処理過程で、スピル操作も行う。スピル付併合は、干渉がない変数同士で行うスピルが出ない併合と、干渉がある変数同士で行うスピルが出る併合の2つのステージに分けられる。

スピルが出ない併合

まず、エッジコスト付干渉グラフ上で、同色で彩色されたノード同士を併合する。これは、エッジコスト付干渉グラフのノードが表すものを、変数から仮想レジスタに変換する操作にあたる。つまり、色を仮想レジスタとみなして、仮想レジスタの生存区間情報と、仮想レジスタ間の干渉度を求めるために行う。我々は、この操作を同色ノードの併合と呼んでいる。

この同色のノードの併合は、エッジが張られていないノード同士の併合なので、以下のような比較的簡単なアルゴリズムで行うことができる。この同色ノードの併合が終了すると、エッジコスト付干渉グラフは、彩色で用いた色数と同数のノードがある完全グラフに変形される。

1. エッジコスト付干渉グラフ内に、同色で彩色されたノードがある限り、以下の操作を行う。
 - (a) エッジコスト付干渉グラフから、同色で彩色された2つのノードを選び、それぞれノードA、ノードBとする。
 - (b) ノードAとの間にエッジが張られているノード集合と、ノードBとの間にエッジが張られているノード集合の、和集合Uを求める。
 - (c) ノードAとノードBの合併後のノードCを作る。
 - (d) ノードAの表す変数の生存区間情報とノードBの表す変数の生存区間情報を、重ね合わせたものをノードCの表す変数の生存区間情報にする。
 - (e) 和集合UのすべてのノードXに対して以下の操作を行う。

- i. ノード X とノード C の間に，双方向に有向エッジを張る．
- ii. ノード A からノード X の干渉度とノード B からノード X への干渉度の和を，ノード C からノード X への干渉度にする．
- iii. ノード X からノード A の干渉度とノード X からノード B への干渉度の和を，ノード X からノード C への干渉度にする．
- iv. ノード A とノード B と，関連するエッジのすべてをエッジコスト付干渉グラフから削除する．

$n \times n$ サイズの符号なし整数の 2 次元配列で実装を行った場合，同色ノードの併合の実装例は，図 5.8 のようになる．この例では，ノード A とノード B を併合している．行 A+B の黒色部分を生成するために行 A と行 B の黒色部分の和を，列 A+B の灰色部分を生成するために列 A と列 B の灰色部分の和をそれぞれ求めている．

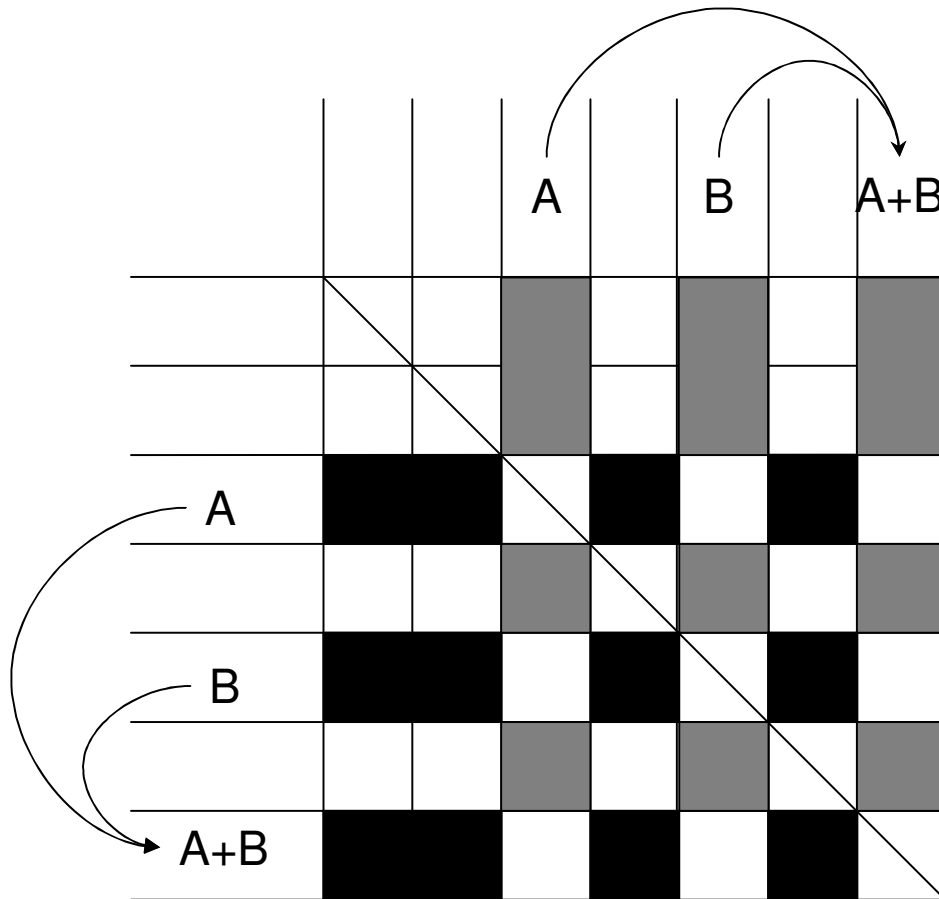


図 5.8: 同色ノードの併合の実装例

図 5.7 のエッジコスト付干渉グラフに，同色ノードの併合を行った例を，図 5.9 に示す．ノード A とノード D，ノード B とノード F が，それぞれ同色塗られているため併合されている．このように，エッジコスト付干渉グラフは 4 つのノードが存在する完全グラフになる．

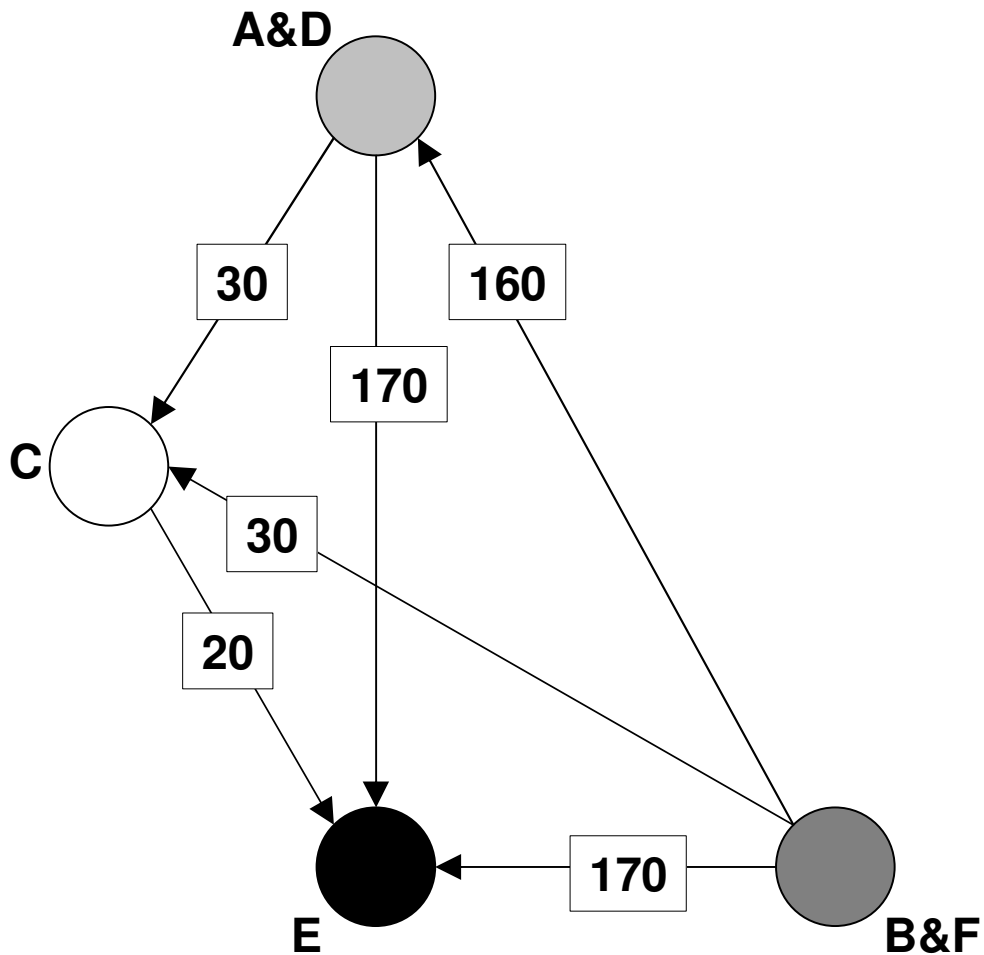


図 5.9: 同色ノードの併合例

スピルが出る併合

同色ノードの併合が終了すると、それぞれのノードが別々の色で塗られたエッジコスト付干渉グラフになっている。ここからさらに、エッジコスト付干渉グラフを過彩色したときに増やした色数と、同数のノードを併合によって減らすことによって、色数をコード発行の対象とするアーキテクチャの物理レジスタ数と同じにする。ただし、干渉がある仮想

レジスタ同士を、強引に同じ物理レジスタに乗せようとしているので、生存区間の重なる部分でスピル操作が必要になる。エッジコスト付干渉グラフでは、エッジコストがスピルにかかるコストとみなせるので、解析的なスピル操作が可能になっている。我々は、この操作を異色ノードの併合と呼んでいる。

この異色ノードの併合は、エッジが張られているノード同士の強引な併合になり、スピル操作を含むため、同色ノードの併合よりは複雑なアルゴリズムになる。そのアルゴリズムの一例を以下に示す。

1. エッジコスト付干渉グラフ内に、コード生成の対象となるアーキテクチャの物理レジスタ数以上のノードがある限り、以下の操作を行う。
 - (a) エッジコストが一番小さいエッジを選択し、エッジの始点にあるノードをノード A、エッジの終点にあるノードをノード B とする。
 - (b) ノード A とノード B の合併後のノード C を作る。
 - (c) ノード C からノード A とノード B を除くすべてのノードと双方向に有向エッジを張る。
 - (d) ノード B の表す仮想レジスタの生存区間の中で、ノード A の表す仮想レジスタの生存区間と重なっている部分に対してスピル操作を行う。
 - (e) ノード A の表す仮想レジスタの生存区間と、ノード B の表す仮想レジスタの生存区間からスピルした区間を除いた生存区間を合わせた生存区間情報を、ノード C の表す仮想レジスタの生存区間情報にする。
 - (f) ノード C から出るエッジが向かうすべてのノード X に対して、以下の操作を行う。
 - i. ノード B の表す仮想レジスタのスピルされた生存区間について、ノード B の表す仮想レジスタとノード X の表す仮想レジスタとの、ノード B の表

- す仮想レジスタを優先的にレジスタに割付けた場合のエッジコストを計算し、ノード B からノード X に向かうエッジのエッジコストから減算する。
- ii. ノード A からノード X に向かうエッジのエッジコストと、ノード B からノード X に向かうエッジのエッジコストの和を、ノード C からノード X に向かうエッジのエッジコストにする。
- (g) ノード C に向かうエッジを持つすべてノード Y に対して、以下の操作を行う。
- i. ノード B の表す仮想レジスタのスピルされた生存区間について、ノード Y の表す仮想レジスタとノード B の表す仮想レジスタとの、ノード Y の表す仮想レジスタを優先的にレジスタに割付けた場合のエッジコストを計算し、ノード Y からノード B に向かうエッジのエッジコストから減算する。
 - ii. ノード Y からノード A に向かうエッジのエッジコストと、ノード Y からノード B に向かうエッジのエッジコストの和を、ノード Y からノード C に向かうエッジのエッジコストにする。
- (h) ノード A とノード B と、関連するエッジのすべてをエッジコスト付干渉グラフから削除する。

図 5.3 の例の場合では、このアルゴリズムを適用すると、まず仮想レジスタ E がスピルされ仮想レジスタ C がレジスタに載るようなコードが出される。次に、仮想レジスタ C の生存が終了する図 5.3 の 5 の命令の後に、仮想レジスタ E のスピルインコードが挿入される。そして、併合後の仮想レジスタ C&E の生存区間情報が、仮想レジスタ C と仮想レジスタ E から生成される。仮想レジスタ C&E の生存区間情報は、図 5.3 の 1 の命令以前から仮想レジスタ E が生存しており、3 の命令から仮想レジスタ C が生存し、5 の命令の次に挿入されたスピルインコードからまた仮想レジスタ E が生存するという情報になっている。

この異色ノードの併合を繰り返し、コード生成の対象となるアーキテクチャのレジスタ

の個数と同数のノードをもつ完全グラフになったときが終了状態である。例の場合では、

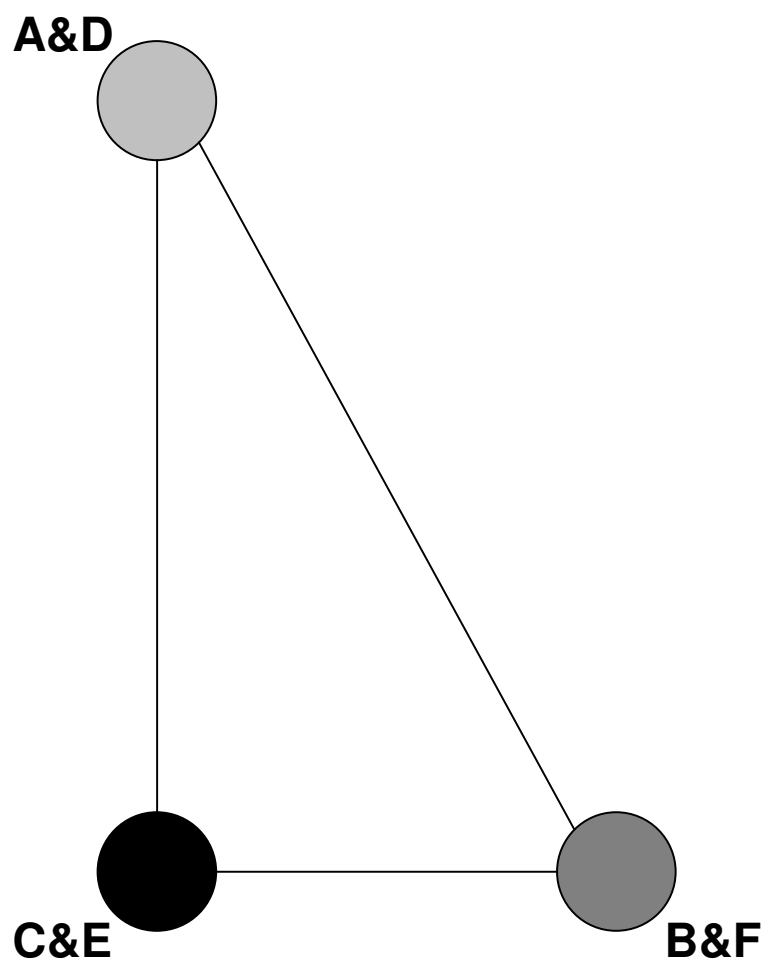


図 5.10: 異色ノード併合の例

5.4 実験

ここでは、性能の比較実験を行う。実験環境は、CPU がインテル Core2Duo プロセッサ 2.66GHz、メモリ 2GB、OS が Windows XP SP2 のマシンを用いた。

本手法を実装するために用いたコンパイラは、Cygwin gcc-3.4.4 という C コンパイラで、レジスタ割付けの部分のみ本手法に書き換えて実験を行った。比較対象としては、Cygwin gcc-3.4.4 のオリジナルを用い、コンパイルにかかった時間と、生成されたコードの実行時間を比較した。評価対象として Cygwin gcc-3.4.4 のオリジナルを選んだ理由は、x86 用のレジスタ割付けにリニアスキャン [15] を用いているからである。

実験に用いたベンチマークは、Stanford Benchmark である。Stanford Benchmark は、Perm、Towers、Queens、Intmm、Mm、Puzzle、Quick、Bubble、Tree、FFT という 10 種類のプログラムから構成されている。各プログラムの説明は以下の通りである。

- Perm：置換プログラム，再帰が多い
- Towers：ハノイの塔の解法プログラム
- Queens：エイトクイーン問題の 50 回解法プログラム
- Intmm：整数行列の乗算
- Mm：実数行列の乗算
- Puzzle：実行時間のほとんどが計算処理となるプログラム
- Quick：クイックソート
- Bubble：バブルソート
- Trees：2 分木ソート
- FFT：高速フーリエ変換

以下で、コンパイル速度の比較と、生成コードの性能比較についてそれぞれ述べる。

5.4.1 コンパイル速度の比較

表 5.2 に、コンパイル 1 回にかかった平均時間をプログラムごとにミリ秒で示す。また、図 5.11 にコンパイル 1 回にかかった平均時間の比率を、プログラムごとにオリジナルを 1 として比較したものを示す。つまり、棒の短い方がコンパイル速度がより速いことを示している。

表 5.2: コンパイルにかかった平均時間 (ミリ秒)

	gcc-3.4.4	本手法
Perm	382	362
Towers	379	359
Queens	381	361
Intmm	380	360
Mm	383	363
Puzzle	379	359
Quick	380	360
Bubble	382	362
Tree	378	358
FFT	381	361

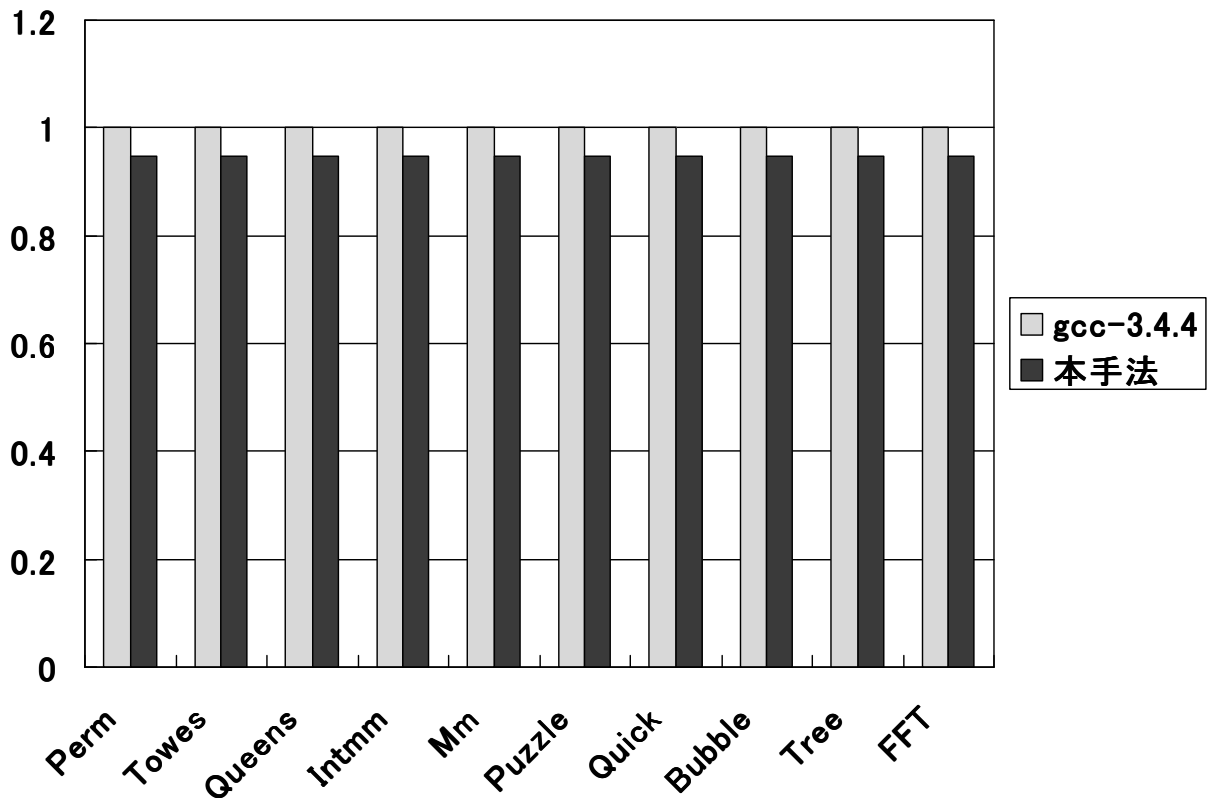


図 5.11: コンパイルにかかった平均時間比率

このように、Cygwin gcc-3.4.4 と比較して、レジスタ割付け時間が短くなった分、コンパイル時間の短縮に成功した。本手法のレジスタ割付け時間が Cygwin gcc-3.4.4 のレジスタ割付け時間より短い理由としては、本手法が全体で 1 回のみレジスタ割付けを行っているのに対して、Cygwin gcc-3.4.4 の用いているリニアスキャンがローカルな変数とグローバルな変数との 2 回のレジスタ割付けを行っている点が挙げられる。

5.4.2 生成コードの性能比較

表 5.3 に、生成されたコードの 1000 回分の実行時間をプログラムごとにミリ秒で示す。また、図 5.12 に生成されたコードの 1000 回分の実行時間の比率を、プログラムごとにオリジナルを 1 として比較したものを示す。つまり、棒の短い方が生成コードの実行速度がより速いことを示している。

表 5.3: 生成コード 1000 回実行の時間 (ミリ秒)

	gcc-3.4.4	本手法
Perm	13293	13016
Towers	7567	7567
Queens	7283	7283
Intmm	2616	2600
Mm	1833	1834
Puzzle	25809	25000
Quick	7310	7033
Bubble	7550	7550
Tree	29685	29417
FFT	5287	4700

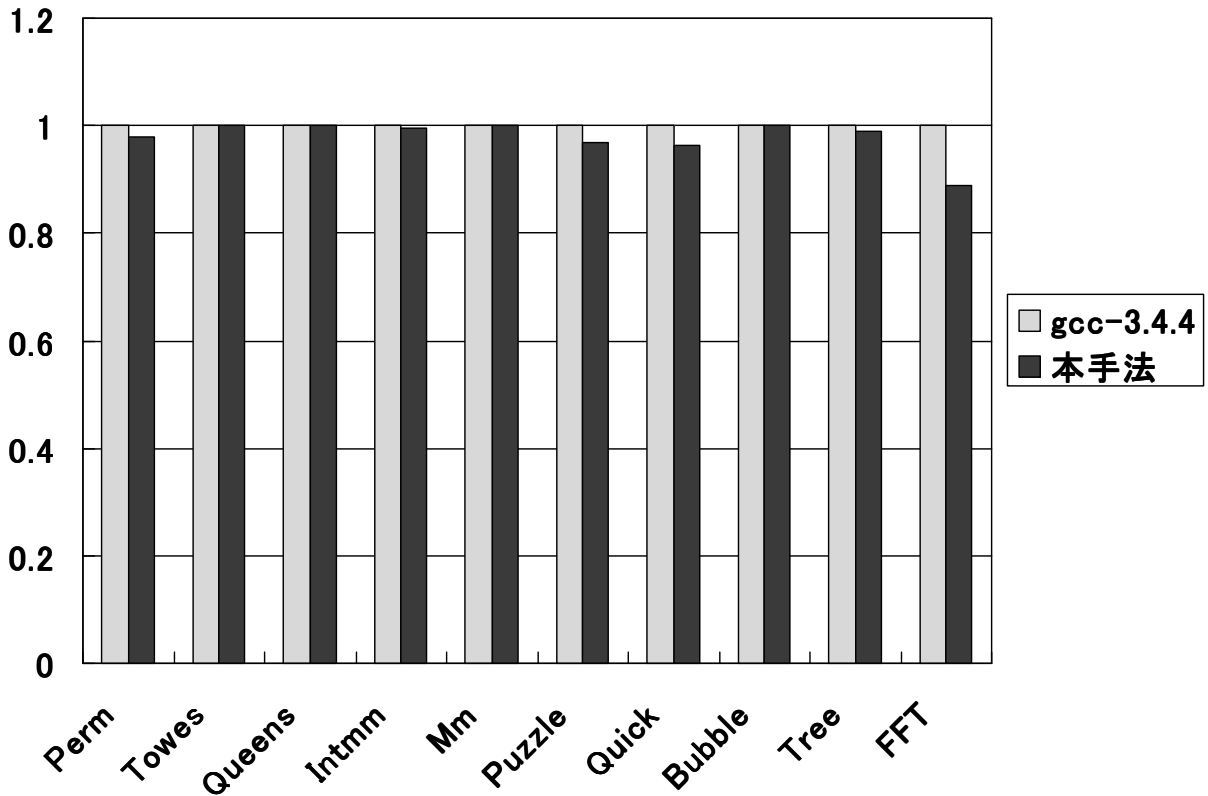


図 5.12: 生成コード 1000 回実行の時間比率

このように、生成されたコードの性能についても良くなっているか、ほぼ同等という結果となった。本手法は、Puzzle のようなループ構造を多く持ったプログラムや、Tree のような末尾再帰があるプログラムに、効果的であるということが言えると思われる。

今回、GCC の RTL から生存区間解析を行っているため、変数の大域的な生存区間分割や、SSA 変換 [4] などは、GCC 任せになっている。また、value numbering[16] といった手法を組み込めば、より高い性能が出る可能性がある。

また、今回の実験は、CPU に複数のコアがあるアーキテクチャ上で行ったが、複数のコアがあることを考慮したレジスタ割付け手法にはなっていない。今後は、複数のコアを

持った CPU が主流になっていくと予測されることから，複数のコアに特化したレジスタ割付け手法も踏まえて拡張していきたいと考えている．

5.5 関連研究

5.5.1 グラフ彩色法

グラフ彩色法 [2] は，最も有名なレジスタ割付け手法である．この手法では，レジスタ割付けを彩色問題に置き換えている．レジスタ割付けも置き換えられた彩色問題も，どちらも NP 完全問題であるため，この問題の最適解を求めようとするとき，非常に長い時間を要する．そこで，静的コンパイラとして許容される時間で解けるように，グラフ理論に基づくヒューリスティクスによる解法を用いている．さらに，アルゴリズムが構造化されているため拡張が容易ということもあって，レジスタ割付けの基本アルゴリズムとして，多くのコンパイラに採用されている．

この手法では，まず変数をノードで表し，生存区間の重なっているノード同士を無向エッジで接続して，干渉グラフと呼ばれるグラフを生成する．次に，コード生成の対象とするアーキテクチャのレジスタ数と同じ色数で，この干渉グラフを彩色することを試みる．もし，グラフを彩色できなかった場合には，メモリに割付ける変数を 1 つ選択し，スピル操作をすることによって，その変数の値をメモリへ退避する．このスピルされた変数の値は，プログラムを高速に実行するために，その値を使用する前にメモリからレジスタに移しておく必要がある．最適化の観点から，メモリからレジスタへ値が移された変数は，メモリへ退避される前の変数とは別の変数と考えることが多い．なぜならば，別々の変数と考えることで，名前依存を考慮する必要がなくなるからである．これによって，プログラム中に表れる変数の数，つまり，干渉グラフのノード数と，それらの間のエッジの数が変更になる．この変更を干渉グラフに反映するために，生存区間解析からの一連の操作を，すべてのノードを彩色できるようになるまで繰り返している．彩色完了後は，色ご

とに物理レジスタを割り当てていくことで，レジスタ割付けが完了する．

この手法では，スピル操作が1度も行われなかった場合には，レジスタ割付けの結果が最適であることが保証される．ただし，スピル操作が行われたときには，静的コンパイラとして許容される時間で解くために組み込まれたヒューリスティクスによって，最適解から離れた結果になることが多い．そのため，スピル操作のヒューリスティクスについては，多くの拡張として手法 [6][7][17] などが提案されている．

しかし，物理レジスタ数の少ないアーキテクチャでは，彩色に使用できる色数が少ないため，スピル操作の回数はどうしても多くなってしまう．言い換えれば，グラフ彩色法を基にしたレジスタ割付け手法では，時間のかかる処理である生存区間解析，グラフ再構築，そして彩色という操作を何回も繰り返さなければならない．また，彩色を試みないと彩色可能かどうかわからず，何回スピル操作を行えばグラフを彩色できるようになるのかわからないという点も問題である．

5.5.2 Optimistic Coalescing

積極的にノード併合を行う手法としては，optimistic coalescing[18] が挙げられる．まず，この手法が提案されるに至る背景から説明する．

ノード併合にも，今まで aggressive coalescing[3]，conservative coalescing[19]，iterated coalescing[20] などさまざまな手法が提案されているが，それぞれ以下のような問題があった．例えば，aggressive coalescing にはノード併合をしすぎて，干渉グラフを彩色できなくなりスピルが発生する可能性があった．その問題を改善した手法として提案された conservative coalescing には，安全な見積もりをしすぎてノード併合を早めにあきらめてしまう問題がある．安全すぎる見積もりを改善した iterated coalescing においても，まだ併合できるノードが残る問題がある．これは，併合したことによって，ノードから出るエッジの数が減ることを考慮していないことに原因がある．

これらの問題を解決した手法が，optimistic coalescing である．この手法では，まず何

も考えずに aggressive coalescing をし、併合されたノードがスピルされることになったら、それを併合前のノードに分解して彩色を行う。また拡張として、コピー文以外のものに対しても併合を行うようにした点は、本手法とも似ている。

しかし、この手法でも併合されていないノードがスピルされた場合には、生存区間解析からやり直す。そのため、物理レジスタ数の少ないアーキテクチャでは、時間のかかる処理を繰り返さなければならない問題は残っている。

5.5.3 リニアスキャン

動的コンパイラ向けの高速なレジスタ割付け手法としては、リニアスキャン [15] が挙げられる。リニアスキャンでは、対象アーキテクチャのレジスタ数の長さを持つアクティブリストと、変数の生存区間情報を持った生存インターバルを利用し、時間ごとにレジスタ割付けを行う。このアクティブリストは、ベーシックブロック内のローカルな変数と、それ以外のグローバルな変数とで、定数個ずつ使用できるようになっている。このアクティブリストに入れる操作が、グラフ彩色法における彩色操作と考えることができる。この際、アクティブリストに入りきらなかった変数について、スピル操作を行っている。このように、スピル操作の際に繰り返しをしない方法になっているため、グラフ彩色法に比べてかなりの速度向上を実現している。

しかし、レジスタ割付けの速度を優先すれば、多くの場合でレジスタ割付けの性能が犠牲になる。この手法では、特に特殊レジスタを使用する変数に対する特別な考慮がないため、特殊レジスタを使用する前後にコピー命令が入ることが多い。また、生存区間の短い変数から積極的にレジスタに割付けるというヒューリスティクスを用いており、生存区間が長くてよく参照される変数がある場合の性能が悪くなっている。この手法が提案された論文 [15] によると、ベンチマークにおける結果で、グラフ彩色法より 10%ほど性能が悪くなっていることがわかる。

5.5.4 リザーブドレジスタ

リニアスキャン同様，動的コンパイラ向けの高速度レジスタ割付け手法の1つにリザーブドレジスタ方式 [21] が挙げられる．この方式では，メモリにスピルされた変数の値を読み込むための専用レジスタを，演算オペランドの最大数だけ確保しておき，それ以外の残ったレジスタの数でグラフ彩色法を行う．グラフ彩色法と大きく異なる点は，この手法では彩色は1度しか行わないという点である．その1度で彩色されなかったノードの表す変数の値については，すべてスピルしてメモリに配置するという方法をとっている．リニアスキャン同様に，スピル操作の際に繰り返しをしない方法になっているため，グラフ彩色法に比べてかなりの速度向上を実現している．

しかし，欠点もリニアスキャンと同様で，レジスタ割付けの速度を優先したことにより，レジスタ割付けの性能が犠牲になっている．この方式では，物理レジスタ数の少ないアーキテクチャがコード生成の対象だった場合に，さらにオペランドの最大数だけ確保するため，彩色に使用できる色数が非常に少ない．このため，干渉グラフのノードをほとんど彩色できず，スピルが頻発してしまうため非常に性能が悪くなってしまふ．つまり，彩色に使用できるレジスタを減らしたことにより，極端なオーバースピルの状態になっているということがいえる．

図 5.13 に，レジスタ数を 4 とし，演算オペランド最大数を 2 として，リザーブドレジスタ方式で彩色した一例を挙げる．つまり，スピルされた変数専用のレジスタが 2 つ必要となるので，2 色で干渉グラフを彩色しようとした結果と捕らえることができる．灰色，黒色で塗られたノードが表す変数はレジスタに割付けられる．しかし，色が塗られなかった E や F のノードが表す変数は，変数の生存区間全域においてメモリに配置され，使用されるときにはスピルされた変数専用のレジスタに載るといったコードが生成される．

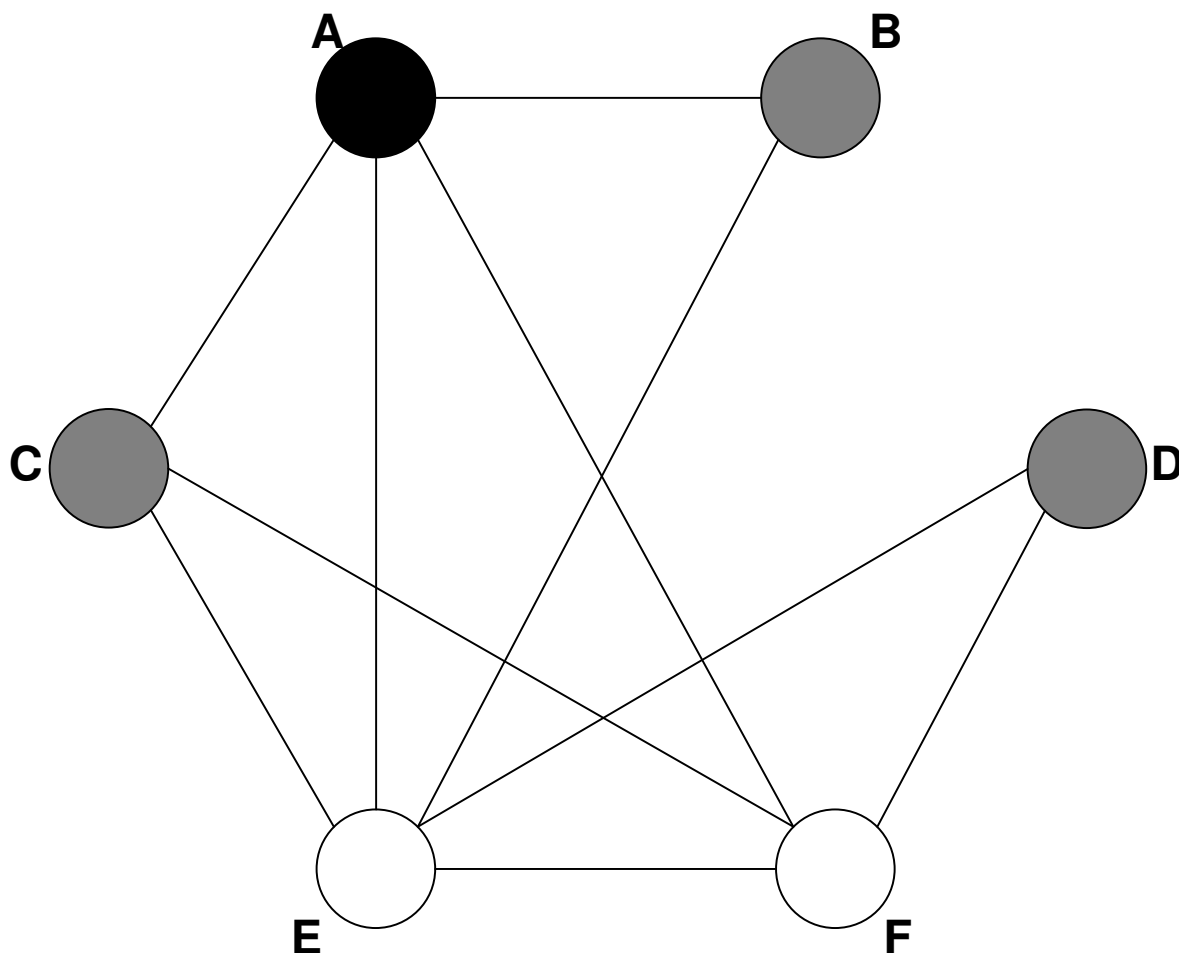


図 5.13: リザーブレジスタ方式での彩色例

5.6 あとがき

本章では、静的コンパイラ向けの、高速でかつ性能もあまり低下しないレジスタ割付け手法を提案した。実験結果では、gcc-3.4.4 より、8%ほどの速度向上と、平均 5%ほどの性能向上が見られた。この実験結果は、gcc-3.4.4 が用いているレジスタ割付け以外の最適化手法を用いた場合の結果である。そのため、gcc-3.4.4 が用いていない最適化との相性

は、今回の実験では検証されていない。どのような最適化手法と協調性があるか、また、どれほどの性能が得られるかどうかは、今後の実験で検証したいと考えている。例えば、生存区間分割において、変数の使用頻度の高い区間と低い区間を、計算量を上げずに切り離すことが可能になれば、今以上の実行性能の向上が見込められると思われる。また、より良い彩色方法の検討もしたいと考えている。

第6章 おわりに

本論文は、コンパイラの最適化において2つの重要な技術である、レジスタ割付けおよびコードスケジューリングにおいて、どのようにすれば実行速度が高速なコードを生成できるのかをまとめたものである。

本論文の2章では、本論文が対象とするアーキテクチャについて述べた。

本論文の3章では、4章と5章で用いるデータ構造の詳細について述べた。

本論文の4章では、プレスケジューリングを用いた命令レベル並列プロセッサ向けのレジスタ割付けおよびコードスケジューリング技法について提案した。本手法は、プレスケジューリングを行うことにより、コンパイル時に想定していた性能より、実行時性能が大幅に下がってしまう可能性がある問題を解決する手法である。評価において、本手法はスケジューリング能力、アルゴリズムの高速性のいずれにおいても、既存の手法に比べて同等以上であることを示した。

本論文の5章では、少レジスタアーキテクチャ向けのレジスタ割付け技法について提案した。本手法は、既存の手法における干渉グラフのノードを併合する段階でスピルを行うことを許すことにより、レジスタ割付けにかかる時間の短縮と、レジスタ割付け結果の向上を図った手法である。レジスタ割付けにかかる時間に関する評価において、本手法は従来手法以上であることを示した。また、レジスタ割付け結果に関する評価において、本手法は従来の高速なレジスタ割付け手法以上であり、従来の低速だが性能のよいレジスタ割付け手法未満であることを示した。

謝辞

本研究は、筆者が早稲田大学大学院在学中および助手在職中になされたものである。本研究を進めるにあたり、終始温かいご指導を賜りました、深澤良彰教授、小松秀昭氏、古関聰氏に感謝の意を表します。

また多くのご助言をいただきました、村岡洋一教授、笠原博徳教授、木村啓二准教授をはじめ、情報理工学科の諸先生方に深く感謝いたします。

さらに、本研究の遂行において、多大なるご助力をいただきました、深澤研究室に在籍された方々、現在在籍されている方々に感謝いたします。

最後に、今日までの筆者の研究生活を支えてくださった家族に感謝いたします。

参考文献

- [1] 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)
- [2] Chaitin, G.J., Auslander, M.A., Chandro, A.K., Cocke, J., Hopkins, M.E., and Markstein, P.W.: “Register Allocation via Coloring”, *Computer Languages*, Vol.#6, pp. 47–57 (1981)
- [3] Chaitin, G.J.: “Register Allocation & Spilling via Graph Coloring” *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pp. 95–105 (1982)
- [4] Cytron, R., Ferrante, J., Rosen, B., Wegman, M. and Zadeck, K.: “An Efficient Method of Computing Static Single Assignment Form”, *Conf. Record of the 16th ACM Symposium on the Principles of Programming Languages*, pp. 25–35 (1989)
- [5] 小松秀昭, 古関聡, 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)
- [6] Norris, C. and Pollock, L.L.: “A Scheduler-Sensitive Global Register Allocation”, *Proc. of the ACM SIGPLAN '93 Conf. on Supercomputing*, pp. 804–813 (1993)

- [7] Pinter, S.S.: “Register Allocation with Instruction Scheduling: a New Approach”, Proc. of the ACM SIGPLAN '93 Conf. on Programming Languages Design and Implementation, pp. 248–257 (1993)
- [8] Berson, D.A., Gupta, R. and Soffa, M.L.: “Resource Spackling: A Framework for Integrating Register Allocation in Local and Global Schedulers”, IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, pp. 135–146 (1994)
- [9] Berson, D.A., Gupta, R. and Soffa, M.L.: “GURRR: A Global Unified Resource Requirements Representation”, Proc. of ACM Workshop on InterMediate Representations, Sigplan Notices, Vol.30, pp. 23–34 (1995)
- [10] 古関聰, 小松秀昭, 百瀬浩之, 深澤良彰: “命令レベル並列アーキテクチャのためのコードスケジューラおよびレジスタアロケータの協調技法”, 情報処理学会論文誌, Vol.38, No.3, pp. 584–594 (1997)
- [11] Inagaki, T., Komatsu, H. and Nakatani, T.: “Integrated prepass scheduling for a Java Just-In-Time compiler on the IA-64 architecture”, Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, pp. 159–168 (2003)
- [12] Knoop, J., Rünthing, O. and Steffen, B.: “Lazy Code Motion”, Proc. of PLDI, Vol.27, No.7, pp. 224–234 (1992)
- [13] Aho, A.V. and Ullman, J.D.: “Principles of Compiler Design” Addison-Wesley, (1977)
- [14] Aho, A.V., Sethi, R. and Ullman, J.D.: “Compilers – Principles, Techniques, and Tools” Addison-Wesley, (1986)

-
- [15] Poletto, M. and Sarkar, V.: “Linear scan register allocation”, ACM TOPLAS, Vol. 21, No. 5, pp. 895–913 (1999)
- [16] Briggs, P.: “Value Numbering” Software: Practice and Experience Vol.27, No.6 pp. 701–724 (1997)
- [17] Bergner, P., Dahl, P., Engebretsen, D. and O’Keefe, M.: “Spill Code Minimization via Interference Region Spilling”, Proc. of the ACM SIGPLAN ’97 Conf. on Programming Language Design and Implementation, pp. 287–295 (1997)
- [18] Park, J. and Moon, S.: “Optimistic register coalescing”, Proc. PACT ’98, pp. 196–204 (1998)
- [19] Briggs, P., Cooper, K.D. and Torczon, L.: “Improvements to graph coloring register allocation” ACM TOPLAS, Vol. 16, No. 3, pp. 428–455 (1994)
- [20] George, L. and Appel, A.W.: “Iterated register coalescing” ACM TOPLAS, Vol. 18, No. 3, pp. 300–324 (1996)
- [21] Koseki, A., Komatsu, H. and Nakatani, T.: “Spill Code Minimization by Spill Code Motion”, 12th International Conf. on Parallel Architectures and Compilation Techniques (PACT’03), pp. 125–134 (2003)

研究業績

論文

非再試行型レジスタ割付けとその評価

片岡正樹, 古関聰, 小松秀昭, 深澤良彰

情報処理学会論文誌: プログラミング, Vol.49, No.SIG1(PRO35), pp.96-102,
January, 2008

レジスタ生存グラフを用いたレジスタ割付けへのプロセッサ並列度の考慮

片岡正樹, 古関聰, 小松秀昭, 深澤良彰

情報処理学会論文誌: プログラミング, Vol.46, No.SIG1(PRO24), pp.10-18, Jan-
uary, 2005

国際会議

Non-Retrial Register Allocation and Its Spill Optimization Method

Masaki Kataoka, Akira Koseki, Hideaki Komatsu, Yoshiaki Fukazawa

The 2007 International Conference on Parallel and Distributed Processing
Techniques and Applications, June 25-28, 2007

研究会

- 生存区間分割時に発生する偽干渉を避けるための同時コピー中間コードの利用
中林淳一郎, 片岡正樹, 古関聰, 小松秀昭, 深澤良彰

2007年並列/分散/協調処理に関する『旭川』サマー・ワークショップ(SWoPP 旭川 2007), August 1-3, 2007, 情処研報 Vol.2007 No.79 pp91-96

- 非再試行型レジスタ割付けとその評価

片岡正樹, 古関聰, 小松秀昭, 深澤良彰

2007年並列/分散/協調処理に関する『旭川』サマー・ワークショップ(SWoPP 旭川 2007), August 1-3, 2007

- 分岐の相関を利用した効率的なパスプロファイリング

野崎晋也, 片岡正樹, 古関聰, 小松秀昭, 深澤良彰

第58回プログラミング研究会 (PRO-2005-5), March 16-17, 2006

- 非反復型レジスタ割付けとそのスピル最適化手法

片岡正樹, 小川健一, 古関聰, 小松秀昭, 深澤良彰

第8回プログラミングおよびプログラミング言語ワークショップ (PPL2006), March 5-7, 2006, PPL2006 論文集 pp163-169

- コンパイル速度の向上を目的とした非反復型レジスタ割付け手法

小川健一, 片岡正樹, 古関聰, 小松秀昭, 深澤良彰

第57回プログラミング研究会 (PRO-2005-4), January 16-17, 2006

- Javaにおける例外処理の実行時情報を利用した最適化

廣澤健, 片岡正樹, 古関聰, 小松秀昭, 深澤良彰

第57回プログラミング研究会 (PRO-2005-4), January 16-17, 2006

- 頻出メソッド管理テーブルを用いた invokeinterface 命令の実行高速化手法

藤本勝平, 片岡正樹, 古関聰, 小松秀昭, 深澤良彰

第57回プログラミング研究会 (PRO-2005-4), January 16-17, 2006

- レジスタ生存グラフを用いたレジスタ割付けへのプロセッサ並列度の考慮

片岡正樹, 古関聰, 小松秀昭, 深澤良彰

第 49 回プログラミング研究会 (PRO-2004-1), May 18-19 ,2004

- ループパーティショニングを用いたベクトル抽出技法

片岡正樹, 佐渡昭彦, 古関聰, 小松秀昭, 深澤良彰

並列/分散/協調処理に関するサマー・ワークショップ (SWoPP2003), August
4-6, 2003

- レジスタ生存グラフを用いたレジスタ割付け及びコードスケジューリング技法

片岡正樹, 古関聰, 小松秀昭, 深澤良彰

並列/分散/協調処理に関するサマー・ワークショップ (SWoPP2002), August
21-23, 2002, 情処研報 Vol.2002 No.81 pp79-84