

平成21年度学士論文

SSA形式を用いたレジスタ割当ての
COINS上での実装

東京工業大学
理学部 情報科学科
学籍番号 04B09611

結束 直道

指導教員
佐々 政孝 教授

2010年2月7日

目次

| | | |
|----------|-------------------------------------|-----------|
| 1 | はじめに | 3 |
| 1.1 | 背景 | 3 |
| 1.2 | レジスタ割当て | 3 |
| 1.3 | 研究概要 | 3 |
| 1.4 | 構成 | 3 |
| 2 | COINS について | 4 |
| 2.1 | COINS について | 4 |
| 2.1.1 | 背景 | 4 |
| 2.1.2 | 構成 | 4 |
| 2.1.3 | COINS のバックエンドについて | 4 |
| 3 | 準備 | 6 |
| 3.1 | 生存区間 (live range) | 6 |
| 3.2 | 区間グラフ (interval graph) | 6 |
| 3.3 | スピル (spill) | 7 |
| 3.4 | 干渉グラフ (interference graph) | 8 |
| 4 | グラフ彩色を用いたレジスタ割当てについて | 9 |
| 4.1 | グラフ彩色 | 9 |
| 4.2 | 既存の彩色手順 | 9 |
| 4.3 | 彩色上の欠点 | 10 |
| 4.4 | SSA 形式の干渉グラフ | 11 |
| 4.4.1 | 弦グラフ (chordal graph) | 11 |
| 4.4.2 | 弦グラフの特徴とその利点 | 12 |
| 5 | SSA 形式プログラムとその干渉グラフ | 14 |
| 5.1 | 基本的な定義 | 14 |
| 6 | SSA 干渉グラフの彩色 | 16 |
| 6.1 | 既存の彩色アルゴリズム | 16 |
| 6.2 | Perfect Elimination Order を用いたアプローチ | 16 |
| 7 | 各フェーズのアルゴリズム | 19 |
| 7.1 | スピル (spilling) | 19 |
| 7.1.1 | 既存のアルゴリズム | 19 |
| 7.1.2 | 今回実装するもの | 19 |
| 7.2 | SSA 逆変換 (SSA-Destruction) | 22 |
| 7.3 | 合併 (Coalescing) | 27 |
| 7.3.1 | 不動点 (fixed point) | 27 |
| 7.3.2 | コスト関数 | 28 |
| 7.3.3 | 不動点を増やすためのアプローチ | 28 |

| | | |
|----------|--------------------|-----------|
| 7.3.4 | アルゴリズム | 28 |
| 8 | 実行結果の評価・考察 | 33 |
| 8.1 | 問題点 | 33 |
| 8.2 | 解決方法 | 33 |
| 8.3 | 実験 | 33 |
| 9 | おわりに | 34 |
| 9.1 | 課題 | 34 |
| 9.1.1 | SSA 逆変換の問題 | 34 |
| 9.1.2 | コスト関数 | 34 |
| 9.2 | 関連研究 | 34 |
| 9.2.1 | 合併に関する研究 | 34 |
| 9.2.2 | SSA 形式の干渉グラフに関する研究 | 34 |

1 はじめに

1.1 背景

プログラム中でメモリにしまわれている値は、レジスタに置き換えることで実行時間を大幅に短縮できることがある。これは、メモリへのアクセスよりもレジスタへのアクセスのほうが時間がはるかに短いからである。メモリにアクセスしていた部分をレジスタへのアクセスのみで済ますことができれば、その分だけ実行時間が改善されることになる。メモリアクセスをレジスタアクセスに置き換えるための最適化の手法の一つとして、レジスタ割当てが挙げられる。レジスタ割当ては、最適化コンパイラの中で最も重要なフェーズの一つで、改善のための様々な研究がなされてきた。

1.2 レジスタ割当て

レジスタを用いた命令はメモリを参照する命令より一般に早く、短い。そこで効率のよいコードをコンパイラにおいて生成するには、変数や一時名(オペランド)をできるだけレジスタに割当て、レジスタを有効に活用することが大切である。レジスタ割当て(Register Allocation)とは、プログラム内の多数の変数を少数のCPUレジスタに保持する技法で、その目標は、プログラムの実行速度を最小化すべく多くのオペランドをレジスタに保持するようにすることである。レジスタ割当てはふつう機械語の目的コードを生成する前か後、場合によっては目的コード生成と同時に行う。

1.3 研究概要

コンパイラの最も重要なフェーズの一つであるレジスタ割当てにおいて、その性能と実行速度の改善のために様々な研究がなされてきた。ここでは、Sebastion Hackらが提唱するSSA形式のプログラムを用いた新たなレジスタ割当てのアーキテクチャ[1]をCOINS上で実装することを試みた。グラフ彩色理論を用いた既存のレジスタ割当てにおけるいくつかの難点を、SSA形式を導入することによって見事に解決することができる。

1.4 構成

本論文の構成を以下に示す。第2章では今回実装を行ったCOINSについての説明を行う。第3章ではレジスタ割付けとその実装のために必要な、生存区間、干渉グラフの知識、基本的なレジスタ割当てに関する知識、及びSSA形式とその干渉グラフの知識を説明する。第4章では、今回用いるアルゴリズムを構成するためのSSA形式とその干渉グラフの関係を説明する。第5章ではSSA形式のプログラムにおける干渉グラフの彩色アルゴリズムについて、第6章では今回実装する新たなレジスタ割当てのアーキテクチャにおけるそれぞれのフェーズで使用しているアルゴリズムの説明を行う。第7章では具体的な実装方法について、第8章ではプロトタイプ実装したものを用いてレジスタ割当てを行い、既存の実装との比較、評価を行う。第9章では結論を述べる。

2 COINS について

本節では、本手法を実装、実験を行う際に利用した並列化コンパイラ向け共通インフラストラクチャ、COINS(compiler infrastructure)[2] について説明を行う。COINS とは、コンパイラ研究の基盤となる共通のコンパイラの作成を目的として研究が進められているコンパイラ・インフラストラクチャである。

2.1 COINS について

COINS とは、コンパイラ研究の基盤となる共通のコンパイラの作成を目的として研究が進められているコンパイラ・インフラストラクチャである。

2.1.1 背景

COINS は、コンパイラ研究の基盤となる共通のコンパイラの作成を目的として 2000 年度より研究が進められている。つまり、組み合わせ可能なコンパイラ部品で構成された共通インフラを作り、その上に各企業や研究者がそれぞれの目的に合う機能部品を加えることができるようにすることを目的としている。COINS では、SSA 最適化を行っている。現在、多くの最適化コンパイラで SSA 最適化の実験が行われているが、本研究室の佐々政孝教授が COINS の研究プロジェクトに携わっているため、本研究室では COINS をインフラとして利用した研究が多く行われており、データの蓄積が行いやすいことなどの理由から、本手法の実装を行うインフラに COINS を選択した。

2.1.2 構成

一般にコンパイラは、フロントエンド(front end) とバックエンド(back end) から構成される。フロントエンドは、原始プログラム(source program) を中間コード(intermediate code) と呼ばれる内部形式に変換する。バックエンドは、中間コードを計算機の機械コード(machine code) に変換する。フロントエンドはさらに、字句解析器(lexical analyzer)、構文解析器(syntax analyzer)、意味解析器(semantic analyzer) に分けられる。バックエンドは、最適化器(optimizer) とコード生成器(code generator) に分けられる。これら各部分は、コンパイラのフェーズと呼ばれる。COINS では、複数の入力言語、複数の対象機種に対応する 2 つの中間コードがある(図 2.1)。入力言語の論理構造に近いレベルの中間コードを、高水準中間表現(high-level intermediate representation, HIR) と呼び、機械語に近いレベルの中間コードを、低水準中間表現(low-level intermediate representation, LIR) と呼ぶ。

2.1.3 COINS のバックエンドについて

バックエンドはフロントエンドが出力した中間コードファイル(LIR) を読み込んで機械語を生成する。COINS バックエンドの処理は、以下のような流れで行われる。

1. Loading

入力の ASCII LIR ファイルを読み込み、内部表現に置き換える。この段階で CFG(Control Flow Graph) 形式への分割も行われる。内部表現は、クラス Module の単一インスタンスにまとめられる。

2. Optimize

ターゲットに依存しない最適化や変換、そのための解析を行う。

3. Instruction Selection

ターゲット CPU の命令を選択する。

4. Register Allocation & Code Scheduling

レジスタ割当てと命令スケジューリング

5. Final

覗き穴最適化などをほどこした後、アセンブラを出力

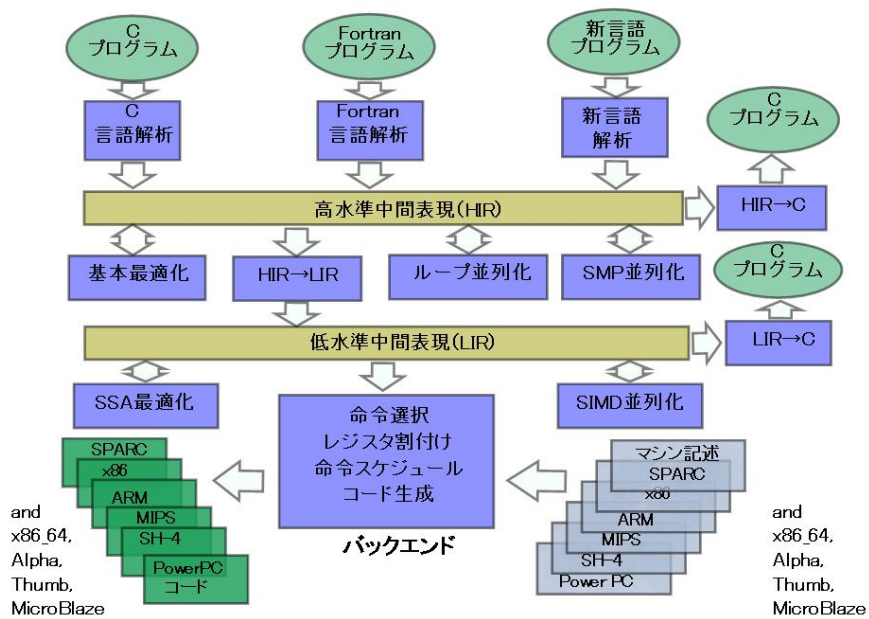


図 1: COINS の構成

3 準備

グラフ彩色理論を用いたレジスタ割当てに関して考察していく前に、必要となってくる概念の解説をここで行う。

3.1 生存区間 (live range)

レジスタ割当ての一般的な方法は、変数の生/死 (live/dead) の情報を使う方法である。変数 x の生存区間 (live range) と変数 y の生存区間に共通部分がなければ、 x と y に同じレジスタを割りつけることができる。このことを利用したレジスタ割当てのアルゴリズムはいくつ存在する。次にこの生存区間を用いて区間グラフについて説明をする。

3.2 区間グラフ (interval graph)

最初に一つの基本ブロックを対象として次のブロックを考えてみる。

```
program P
  1: a = 1
  2: b = a + 1
  3: c = a * b
  4: d = c / b
  5: write(c,d)
```

左端の番号順に実行される。この番号をステップ番号とよぶことにする。同時に発行できる命令数が1であるマシンでは、命令の並んでいる順番に付けられた番号である。以下ではそのようなマシンを仮定する。

1 ~ 5 のステップ番号からなる基本ブロック以外では、すべての変数が死んでいるとすれば、生存区間は以下ようになる。

a: [1,3)
b: [2,4)
c: [3,5)
d: [4,5)

(ここでは $[s,t)$ ステップ s からステップ t の直前までが生存区間を表す。) たとえば、ステップ1で定義された a はステップ3で使われるのが最後なので、ステップ3で定義される c と同じレジスタに入れることができる。このことから、ステップ3に対してレジスタを割り当てるときには a は死んでいると考えてよいことがわかる。これが、 a の生存区間をステップ3の直前までとしている理由である。この生存区間は図2のように表現することができる。図2における横軸は命令ステップ番号を表す。

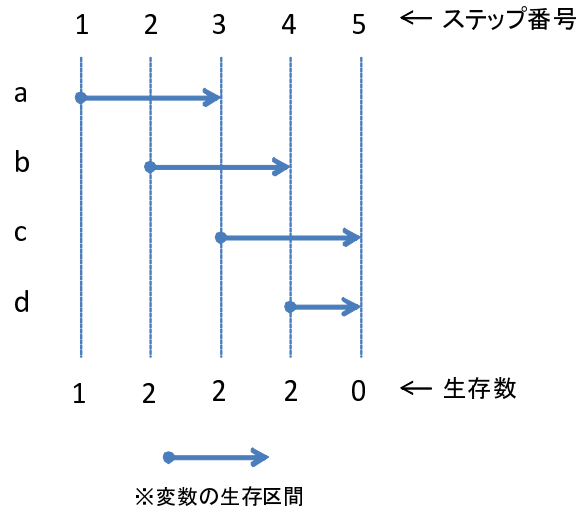


図 2: 区間グラフ

横軸はここでは時間軸と考えることもできるが、レジスタ割当ては時間に対してというより、命令に対して行うものであるので、命令ステップ番号とした。同時に複数個の命令を発行できるマシンでは、同じステップ番号のところに複数個の命令があることになる。図2でたとえば a と b の生存区間には重なりがあり、同じレジスタに割り当ててはできない。a と c の生存区間には重なりがない。

定義 3.1 区間グラフ G のあるステップ p で生存区間が n 重に重なっている時、 p での生存区間、あるいはグラフの幅が n であるといい、 $n = width(G, p)$ と書く。

定義 3.2 区間グラフ G のすべてのステップ p での生存数の最大値 $maxwidth(G, p)$ を $W_{max}(G)$ と書き、最小値 $minwidth(G, p)$ を $W_{min}(G)$ と書く。

生存数が n の点では、レジスタは n 個必要である。従って、基本ブロック B から作られる区間グラフを G_B としたとき、ブロック B 全体ではレジスタは少なくとも $m = W_{max}(G_B)$ 個必要である。なぜならブロックの先頭から順次あいているレジスタに生存区間を割り当てていき、生存区間が終わったところでレジスタを空きレジスタに戻していけば、使用しているレジスタは常に m 以下になるからである。

図2の例では、生存数の最大値が2であるから、レジスタは2つあればよい。たとえば、a と c をレジスタ番号1、b と d をレジスタ番号2に割り当てることができる。

3.3 スピル (spill)

区間グラフ G について、 $m = W_{max}(G)$ を満たしているとき、もし実際のレジスタ数が m 個なければ、ある生存区間には作業用レジスタを使ったり、あるレジスタを選んでその値

を退避することなどが必要になる。これはレジスタの生存区間を短く分けることに相当し、レジスタの追い出し (register spill) と言われる。略してスピル (spill) と言う。スピルが起こった場合は、レジスタをメモリに保存/再度読み込み (save/load) をする命令をプログラム中に挿入する必要がある。

3.4 干渉グラフ (interference graph)

レジスタ割当てに彩色アルゴリズムを用いる際に、生存区間の干渉グラフ (interference graph) を用いる。生存区間の干渉グラフとは、各生存区間を節 (ノード) とし、生存区間に重なりのある二つの生存区間を辺 (edge) (無向辺) で結んだグラフである。生存区間に重なりのある二つの生存変数には同じレジスタを割り当てることができないので、その二つの変数の生存区間はお互いに干渉するという。

プログラム P から得られる生存区間 (図 2) をもとに、干渉グラフで表したものが図 3 となる。

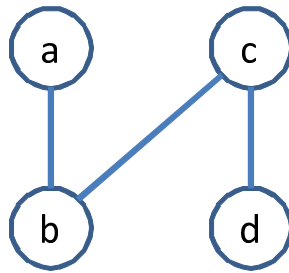


図 3: P の干渉グラフ

ノード間が辺で結ばれていたら、そのノードが表す二変数は生存区間に重なりがあることを示すので、同じレジスタに二変数を割り当てることはできない。たとえば図 3 で考えると、(a,b)、(b,c)、(c,d) のそれぞれの二変数の組に対して、同じレジスタを割り当てることはできない。

4 グラフ彩色を用いたレジスタ割当てについて

レジスタ割当ては、最適化コンパイラにおいて最も重要なフェーズの一つである。メモリよりもアクセス速度の速いレジスタを有効活用することによって、プログラム全体の実行速度を速くすることができるからである。現在まで様々な研究がなされてきたが、中でもグラフ彩色を用いたものは成功したアプローチである。

4.1 グラフ彩色

グラフ彩色理論のレジスタ割当てへの有用性には、プログラム中のそれぞれの変数がグラフ中のノードとして配置されるという、グラフ自身の単純さが貢献している。このグラフを干渉グラフ (interference graph) という。同じレジスタに収めることができない二つのレジスタをコンパイラが見つけたら、それらの変数を示す干渉グラフ中の二つのノード間に辺を結ぶ。干渉グラフ中の隣接する2つのノードが同色にならないように k 色で彩色することが、最大 k のレジスタを使用したレジスタ割当てを表す。

4.2 既存の彩色手順

既存のグラフ彩色を用いたレジスタ割当ては、以下の図のように行う。ここでは使用可能なレジスタ数を k として考える。

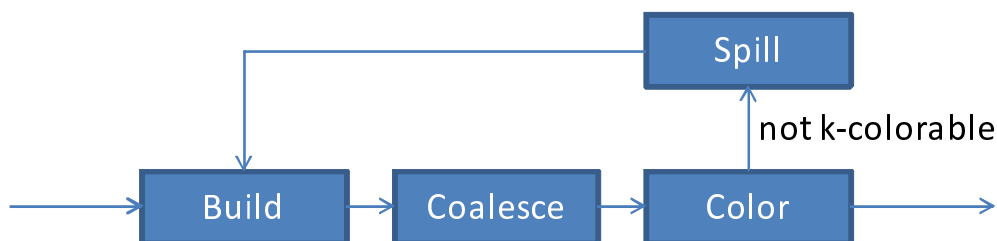


図 4: 既存のグラフ彩色

それぞれのフェーズにおける動作を簡単に説明していく。

Build 全ての変数の生存区間から干渉グラフを構築する。

Coalesce コピー命令における変数に対して、同じレジスタを割り当てることによってコピー命令を不要とすることができる場合がある。この方法を合併 (coalesce) と呼ぶ。

Color 干渉グラフに対して、彩色アルゴリズムに基づき彩色を行っていく。

Spill Color のフェーズで得られた彩色数がレジスタ数を超過してしまった場合、レジスタを選んで値を退避する。

以上の作業を、 k 色で彩色できるまで繰り返し行う。

4.3 彩色上の欠点

しかし、干渉グラフの彩色数を求める問題は NP 困難であり、さらに以下の 2 つの問題点がある。

- 合併 (coalesce) がグラフの彩色数を増やす可能性がある
- スpill (spill) がグラフの彩色数を改善するかどうかわからない

以上の二つによって、与えられたレジスタ数で彩色可能かどうかを調べるために、再び干渉グラフを構築し、彩色しなおすという作業を繰り返し行う可能性がある。non-SSA 形式の干渉グラフは大きなデータ構造であるため、構築は容易ではない。そのためプログラムの実行時間に少なからず影響を及ぼしてしまう。

たとえば、以下の図 5 のプログラムについて考えてみる。このプログラムに対して、干渉グラフを構築したものが図 6 となる。このグラフに対して彩色を行うと、最適な彩色数は 2 となる。このことを踏まえて、合併を行ってみる。

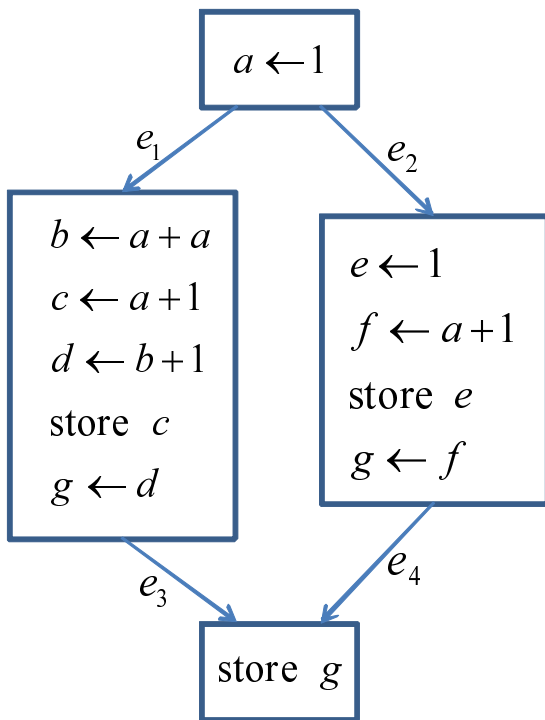


図 5: プログラム P

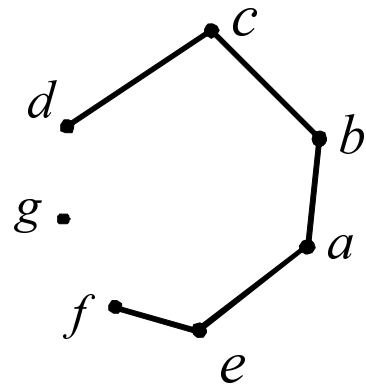


図 6: P の干渉グラフ

このプログラムに対し、積極的な合併 (aggressive coalescing) を行うと、ノード d、g、f は一つのノードに融合される。以下の図 7 が合併後の干渉グラフを表す。

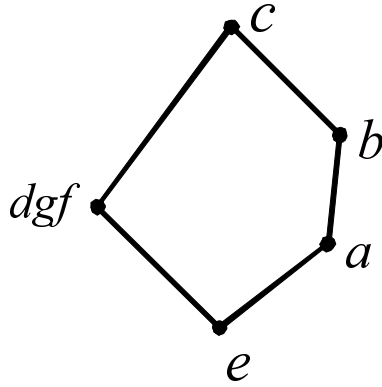


図 7: P の合併後の干渉グラフ

合併後の干渉グラフに対して彩色を行うと、最適な彩色数が 3 に増えてしまっていることがわかる。このことは、コピー文を除去することによって（合併を行うことによって）、プログラム P におけるレジスタの必要数を増やしてしまうことを表している。これでは、コードの改善のために行う合併がより実行時間のかかるコードにしてしまう可能性がある。

次にスピルについて述べる。ノードのスピルがグラフの彩色数を改善するかわからないということは、すなわちスピルによって再構築されたプログラムを調べなおさなければならないということを表す。彩色数がレジスタ数以下になるまで、干渉グラフを再構築し、彩色しなおさなければならないのである。レジスタ数が少なければ少ないほどスピルの数が増えるためこの繰り返しが多くなり、この繰り返しが大きなコストにつながるのである。（干渉グラフは大きなデータ構造であるため、構築は容易でない。）

既存のグラフ彩色にはこのような大きな難点があった。しかし、SSA 形式を導入することによってこれらの問題をきれいに解決することができる。

4.4 SSA 形式の干渉グラフ

SSA 形式の干渉グラフは後に述べるように弦グラフ (chordal graph) である。

4.4.1 弦グラフ (chordal graph)

SSA 形式の干渉グラフが弦グラフであることによる利点を述べる前に、弦グラフの定義や特徴について述べる。

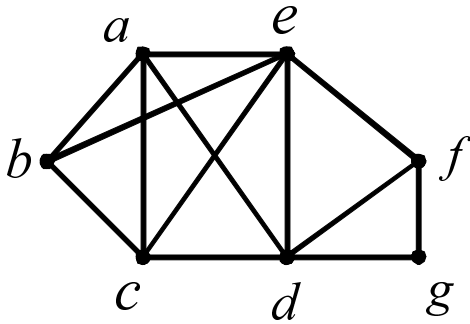


図 8: 弦グラフの例

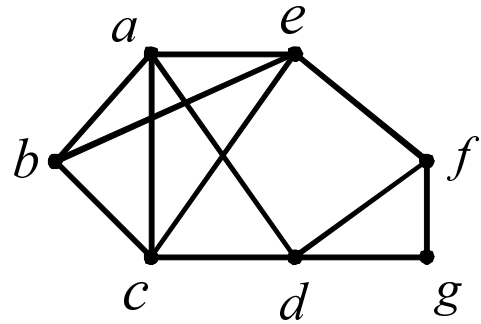


図 9: 弦グラフでないもの

図 8 について、例えば頂点 a, b, c, d, e が、それぞれ辺 $\{a, b\}, \{b, c\}, \{c, d\}, \{d, e\}, \{e, a\}$ でつながっている場合、これをサイクルと呼ぶ。このとき、 $\{a, c\}, \{a, d\}, \{b, e\}, \{c, e\}$ のような、サイクル上にあり、サイクルを構成しない辺のことを弦と呼ぶ。以上を用いて、弦グラフは以下のように定義される。

定義 4.1 グラフ中の長さ 4 以上のサイクルはすべて弦を持つようなグラフを弦グラフ (chordal graph) という。

たとえば図 8 を見てみると、長さ 4 以上のサイクルはすべて弦を 1 つ以上持っている。よってこのグラフは弦グラフとなる。一方図 9 について、サイクル c, d, f, e に注目してみる。このサイクルは長さ 4 であるが、弦を持っていない。よってこの図 9 のグラフは弦グラフではないといえる。

4.4.2 弦グラフの特徴とその利点

まず、弦グラフの特徴を述べる前に以下を定義する。

定義 4.2 グラフ中のある頂点集合 C を見たとき、どの二つのノード間も辺で結ばれているなら、 C をクリーク (clique) と呼ぶ。

例えば、図 8 中の (a, c, d, e) がクリークである。クリークを彩色しようとした場合、すべてのノード間に干渉があるので、同じ色に彩色できるノードの組はない。よってクリークの彩色数はクリークのノードの数と一致する。

弦グラフの大きな特徴として以下の 2 つが挙げられる。

- 弦グラフの彩色数はグラフ中の最大クリークの大きさ (節の数) に等しい。
- 弦グラフは節数の二次の時間で最適に彩色できる。

さらに、既存のグラフ彩色レジスタ割当ての難点を回避できるような、SSA形式のプログラムとその干渉グラフのいくつかの関係がある。

- 干渉グラフ中のクリークは、プログラム中の同時に生存する変数のセットと一致する。クリークの大きさを調べるということは、プログラムに対していくつのレジスタが必要になるかを調べることと同値である。(干渉グラフを構築し、彩色をすることと同値と言える。)プログラム中のそれぞれのステップで生存変数をk個まで減らすことができれば、グラフはk色で彩色できる。
- SSA形式のプログラムにおける変数の支配木を用いることによって、最適な彩色の順番を容易に得ることができる。
- ここで用いるアルゴリズムでは、不要なコピー分の消去を、彩色法の改善によって行う。すなわちグラフの構造を変えるものではないので、彩色数を保つことができる。よって、不要なスピルが必要になることはない。

これらのアプローチを用いていくことによって、下の図のようなループのないレジスタ割当ての構造に導くことができる。



図 10: 新たなグラフ彩色のアーキテクチャ

5 SSA 形式プログラムとその干渉グラフ

SSA 形式を用いた具体的なアルゴリズムに入る前に、SSA 形式プログラムとその干渉グラフの基本的な性質を解説する。このことは、後のアルゴリズムで利用していくこととなる。

5.1 基本的な定義

まず、以下を定義する。

定義 5.1 (ラベル) プログラム中のそれぞれの命令に付けられた識別子をラベルとする。

定義 5.2 (D_v) 変数 v が定義されたステップのラベルを示す。

SSA 形式のプログラムにおいては変数定義が唯一であることをふまえると、この D_v も同様唯一でなければならない。また、プログラムの開始時の最初のラベルを $start$ とする。

定義 5.3 (支配 (dominance)) $start$ からラベル l' までのパスが必ずラベル l を含むとき、 l は l' を支配するといい、

$$l \preceq l' \quad (1)$$

と書く。

定義 5.4 (Strict Program) プログラム上のそれぞれの変数 v の使用が D_v に支配されているとき、このようなプログラムをストリクトであるという。

ここで、SSA 形式プログラムの干渉グラフについて、プログラム P の干渉グラフは以下の情報で成り立つ。

$$G = (V, E)$$

V: ノードの集合
E: 辺の集合

定義 5.5 (干渉 (interference)) プログラム中のある変数 v と w がともに生存するようなラベルがあるとき、2 変数は干渉するという。

また、Buldimlic は同じ論文の中で以下の 2 つの lemma を与えている。

系 5.1 変数 v と w が干渉しているとき、

$$D_v \preceq D_w$$

または

$$D_w \preceq D_v$$

である。

系 5.2 変数 v と w が干渉し、かつ $D_v \preceq D_w$ であるとき、変数 v は D_w で生存する。

以上の Buldimlic の系を用いて、S.Hack らは以下の定理を証明している。

定理 5.3 SSA 形式プログラムの干渉グラフ $G = (V, E)$ において、すべてのクリーク $C = c_1, \dots, c_n \subseteq V$ について、 $c_1 \sim c_n$ がすべて生存するようなラベル $l \in \text{labels in } P$ が存在する。

また、SSA 形式のプログラムにおいて、Hack らは Lengauer らの論文 [11] から以下のことを証明している。[1]

定義 5.6 SSA 形式のプログラムにおける干渉グラフは、弦グラフになる。

6 SSA 干渉グラフの彩色

この節では、SSA 形式のプログラムにおける干渉グラフの彩色アルゴリズムを述べる。

6.1 既存の彩色アルゴリズム

まずは非 SSA 形式プログラムの干渉グラフに対しての、既存の彩色方法を述べる。与えられた干渉グラフに対して、割当てに使用することができるレジスタ数が N であるとする。これは干渉グラフを N 色で彩色する問題になる。

あるノード x と辺で結ばれているノード（これを x の隣接ノード (neighbor node) と呼ぶ) の数が N より小さければ、それらの隣接ノードにどのようなレジスタ番号 (色) が割当てられていても、 x にはそのどれとも違う番号を割当てることができる。したがってもとの干渉グラフから x を除いた部分グラフが N 色で彩色できれば、 x を加えても結果 N 色で彩色できる。このことを利用したアルゴリズムは以下ようになる。

定義 6.1 (度数 (degree)) 一般的に干渉グラフについて、節から出ている枝の数を度数 (*degree*) といい、 x の隣接ノードの数を $\text{deg}(x)$ と表す。

```
生存区間から干渉グラフ  $G$  を作る
spill_list を空にする
while ( $G$  は空でない) do
    if ( $\text{deg}(x) < N$  となる  $x$  が存在する)
         $x$  (および  $x$  からのすべての辺) を  $G$  から取り除く
    else
         $G$  のノード  $x$  を選び、 $x$  を spill_list に加える
    end if
end while
if (spill_list が空)
    取り除いた順序の逆の順にノードを彩色する
else
    spill_list の各ノード  $x$  の生存区間を分割する
    このアルゴリズムの先頭に戻る
end if
```

6.2 Perfect Elimination Order を用いたアプローチ

ここで、彩色する順番 (アルゴリズム中の x を G から取り除くフェーズ) について考えていく。グラフに対して最適に彩色する順番が存在することはグラフ理論では周知であるが、グラフ彩色が NP 困難であること同様に、彩色する順番の決定も NP 困難である。そこで、以下のアプローチで行っていく。

- 最新のグラフ（干渉グラフのノードを取り除いた後に残る部分グラフ）において、隣接ノードがクリークを形成するようなノードを取り除く。

この手法を、Perfect Elimination Order (以後 PEO) という。
PEO について、以下に例を示す。

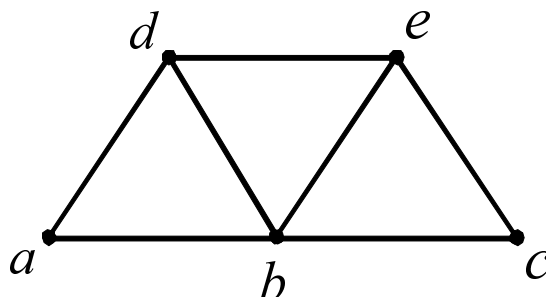


図 11:

図 11 に対して、PEO を適用してみる。PEO にあてはまるような順番はいくつかあるが、例えば $\{a,d,b,c,e\}$ が PEO である。逆に $\{b,a,d,e,c\}$ は PEO ではない。ノード b を取り除く際、隣接ノードである $\{a,d,e,c\}$ がクリークを形成していないからである。

もちろんすべてのグラフに PEO が存在するわけではない。たとえば、図 12 のようなダイヤモンドグラフ (diamond graph) と呼ばれるものがよく知られている。グラフ中のすべてのノードにおいて隣接ノードがクリークを形成しないので、PEO を適用することができない。

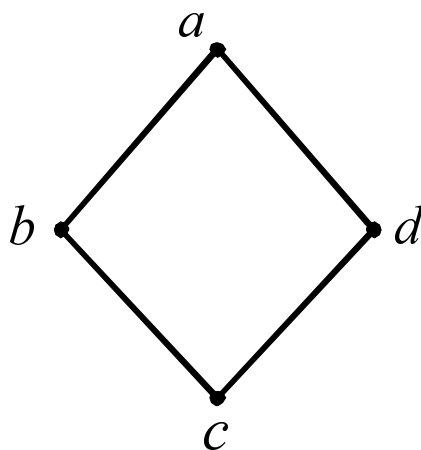


図 12: ダイヤモンドグラフ

この PEO に関して以下のことが知られている。[10]

- 弦グラフにおいては、必ず PEO が存在する。
- グラフに対して PEO を適用して彩色すれば、最適に彩色を行うことができる。

HackらはBudimlicの補助定理を用いて、SSA形式プログラムの変数の支配関係から干渉グラフのPEOを容易に導けることを示している。

定理 6.1 変数の定義が v によって支配されているようなすべての変数が PEO によってすでに取り除かれている場合、 v も PEO としてグラフから取り除くことができる。

すなわち、SSA形式プログラムにおける干渉グラフのPEOは、プログラムの支配木を後順で渡り歩くことで簡単に得られる。また、支配木のみを用いるので、干渉グラフが必要なく、構築する必要もないのである。以上のことから、節の数の2次の時間でSSA形式のプログラムの干渉グラフを彩色できるのである。

弦グラフにおいてPEOが必ず存在するということが一般的に知られている。弦グラフは理想グラフ (perfect graph) に属するため、理想グラフの特徴が弦グラフにも同様にあてはまる。

定義 6.2 (perfect graph) グラフ G について、 G におけるすべての導出部分グラフにおいて最大クリークの大きさと彩色数が一致するとき、 G を理想グラフという。

7 各フェーズのアルゴリズム

7.1 スピル (spilling)

すでに説明したように、干渉グラフの彩色数は一番大きいクリークの大きさによって決まる。また、定理 (5.3) より

- 干渉グラフはそれぞれのクリークにおいて、そのクリーク中のすべての変数が生存するようなラベルが必ず存在する。

このことから、それぞれのラベルにおいて生存変数の数を k 以下に減らすことと、その干渉グラフの最大クリークの大きさを k 以下にすることは同値であることが分かる。任意の干渉グラフの彩色数を求めることに比べ、それぞれのラベルにおける生存変数の数を求めることは干渉グラフの構築の必要がないなど、容易に調べることができる。すべてのラベルにおいて生存変数の数を k 以下にすることができれば、スピル後の干渉グラフは必ず k 彩色可能であるので、レジスタ割当てにおいて彩色問題とスピル問題を分割して考えることができる。

7.1.1 既存のアルゴリズム

伝統的なレジスタ割当てでは、彩色失敗時までスピルは行われず。すなわち、スピルを行うかの決断とグラフの彩色方法が強く関係する。彩色アルゴリズムの中で、ノードが彩色される際、その隣接ノードが使用可能な色を使い尽くしてしまっている場合、スピルのためにその隣接ノードがマークされる。要するに、ノードのスピルは彩色できないという理由によってのみ行われるのだ。このことによって、以下の情報を隠すことになってしまう。

- 変数がどれくらい使用されているか (頻度)
- 変数がどこで使用されているか (場所)
- 次の変数の使用がどのステップなのか (次使用の場所)

伝統的なレジスタ割当てにおけるスピルでは、 k 色で彩色可能なグラフにする目的で、グラフの構造を改善することにのみ目を向けていた。そしてこのレジスタ割当てをより良いものにするために様々な研究がおこなわれてきた。(Bergner[5], Chow and Hennessy[6])

7.1.2 今回実装するもの

しかし、Hsu[7] のように基本ブロック指向のスピル方法などを織り交ぜることによってより良いアルゴリズムができる。Guo[8] は論文で Belady の 最小化アルゴリズム (Belady's MIN Algorithm[9]) の効果を述べている。このアルゴリズムはロードやストアの回数を最小にするものではないが、それでもなお大きな効果が得られる。

[Belady's MIN Algorithm]

主体となる原理は、ステップ番号に注目し将来使用するのが一番後になるような変数をスピルすることである。使用可能なレジスタ数を超える生存変数があった場合、変数中次使用が一番遠いものを選び、スピルする。そしてスピルされた変数の使用があるステップに到達した場合、変数をリロードする。この際に変数のあふれが生じた場合、先ほどと同様にスピルを行う。

例として以下のような場合を考えてみる。

レジスタ数: 4

レジスタの内わけ: { a, b, c, d }

$$l: f \leftarrow \tau(a, e)$$

ラベル l において関数 τ の引数である e をレジスタに割当てて必要がある。しかし、使用可能なレジスタの数が 4 で、現在 { a, b, c, d } と割当てられている。ラベル l での生存変数は { a, b, c, d, e } なので、生存変数の数が使用可能レジスタ数 4 を超えてしまっている。よってレジスタに空きを作るためにスピルを行わなければならない。a は関数 τ の引数としてレジスタに保持しておく必要があるため、残りの変数 b, c, d の中からスピル対象となる変数を選ぶ。基準は上で述べているように、次使用が一番後になる変数を選ぶ。

ここで以下の 2 つの問題が生じる。

1. v の次使用がどれほど後になるのか

スピル対象の変数を選ぶためにも、それぞれの変数の次使用がいつなのかを知る必要がある。しかし、現在注目している基本ブロックの出口では生存し、そのブロックでの使用が以後ないものに関しては、コントロールフローによって使用のタイミングに差が出てくるものがある。しかしどのコントロールフローで使用されるかを確定することはできないので、ここでは次使用が複数のパスの中で一番近いものを選び、それを判断の基準とする。

2. それぞれの基本ブロックの入り口におけるレジスタの内訳
プログラム全体の変数の集合を I とし、以下を定義する。

[定義] I_B : 基本ブロック B に入る際生存しているすべての変数と、 B 中のすべての ϕ 関数の結果が入る変数で構成される I の部分集合

もし、 $|I_B| > k$ の場合、 I_B から次使用が近いものを k 個選ぶ。さらに、使用前にスピルされる I_B 中の変数 v がある場合、ブロックのエントリーで v をレジスタに保持する

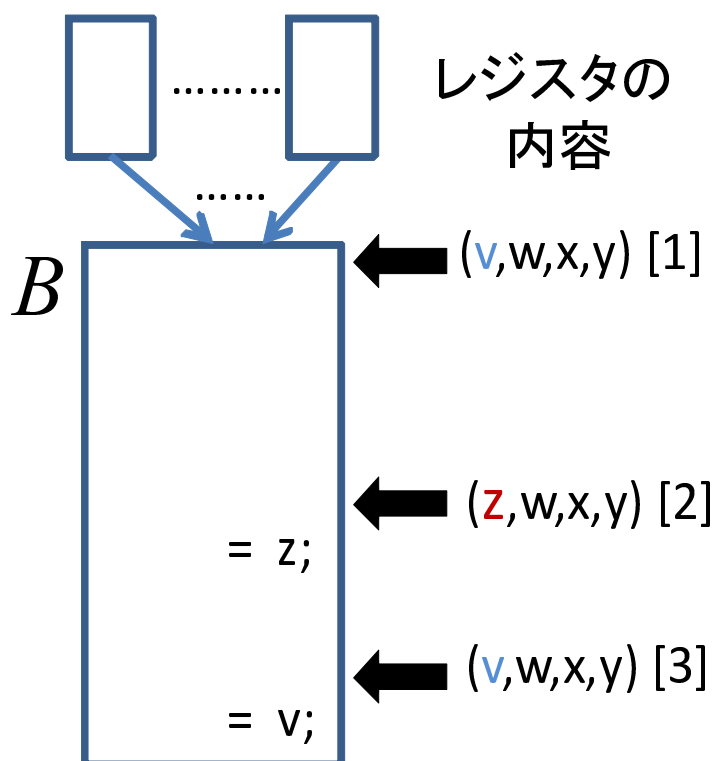


図 13: レジスタ内容の流れ

ことはレジスタの使用効率を悪くする。基本ブロック内での使用を見越してレジスタに保持したとしても、結局使用前にスピルしてしまうからである。よって、 I_B から v を外す。

2 の具体例を図 13 を用いて解説する。

ブロック B における v の使用のためにブロックの入り口 [1] で v をレジスタにおくのだが、もし v の使用 [3] の前において v のスピルが予定されていたら [2]、変数 v に対して unnecessary ストア、ロードを行うことになってしまう。よって変数が使用される前にブロック内でスピルされるような場合、ブロックの初めではレジスタに置いておかない。

以上の基準でスピル対象の変数を決めていき、基本ブロック内の最後のステップの後、レジスタの占領を O_B に記録する。

そして、最後にプログラム中のすべての基本ブロックに適用されたアルゴリズムの結果を合わせ、全体の作業の結果を出す。レジスタは k より多くの占領を許さないなので、あるブロック B に向かうそれぞれのコントロールフローエッジで、 I_B 中の変数がすべてレジスタにあることを約束しなければならない。ここで、 B へのコントロールフローを持つブロックを P とすると、 I_B にあり O_P にない変数があれば、 P から B へのコントロールフロー上でその変数をすべてリロードしなければならない。

7.2 SSA 逆変換 (SSA-Destruction)

SSA 形式には、 ϕ 関数という特有の関数が存在する。

$$\phi \text{ 関数} : y = \phi(x_1, x_2, \dots, x_n)$$

この関数の持つ意味は、 ϕ 関数のラベルに i 番目のコントロールフローを經由してこのブロックに入ってきた場合、

$$y = x_i$$

のようにコピー文と同様の動作をする。

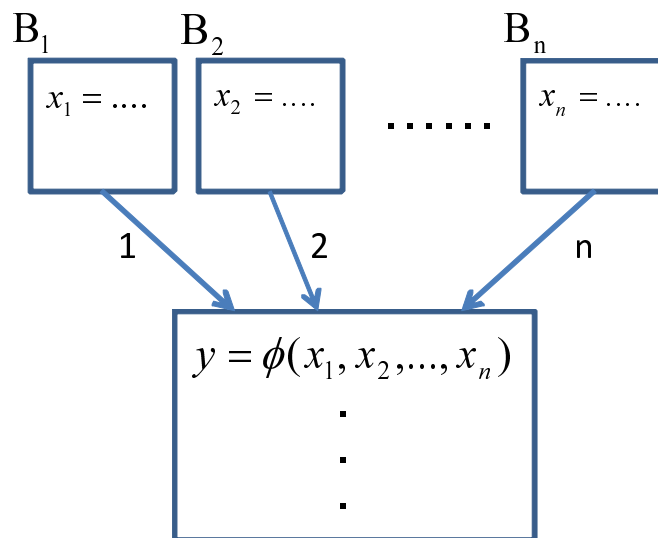


図 14: ϕ 関数

例えば図 14 において、ブロック B_1 から ϕ 関数に入ってきた場合、 y には x_1 の値が入る。ブロック B_2 から ϕ 関数に入ってきた場合、 y には x_2 の値が入る。このようにして、 ϕ 関数のあるブロックにどのコントロールフローから入ってくるかによって、 ϕ 関数の引数が選ばれる。

また、基本ブロックに複数の ϕ 関数がある場合、この ϕ 関数は他のすべての命令の前に同時に実行されるという特徴がある。

$$y_1 = \phi(x_{11}, \dots, x_{1n})$$

$$y_2 = \phi(x_{21}, \dots, x_{2n})$$

.....

$$y_m = \phi(x_{m1}, \dots, x_{mn})$$

例えば以上のように基本ブロック内での ϕ 関数が置かれていたとする。もし、 j 番目のコントロールフローからこのブロックに入ってきたとすると、この ϕ 関数は、すべての i ($i = 1, 2, \dots, m$) に対して一度に x_{ij} を y_i にコピーするバルクコピー (bulk copy) として動作する。

$$\begin{array}{ll}
 y_1 & x_{1j} \\
 y_2 & x_{2j} \\
 & \dots \\
 y_m & x_{mj}
 \end{array}$$

以上の特徴に基づき、伝統的な SSA 逆変換では、 ϕ 関数は単にコピー文と置き換える。しかしこの種類の ϕ の除去方法は、不必要にレジスタの要求を増やす可能性がある。図 15 のプログラム P を用いてその例を示す。

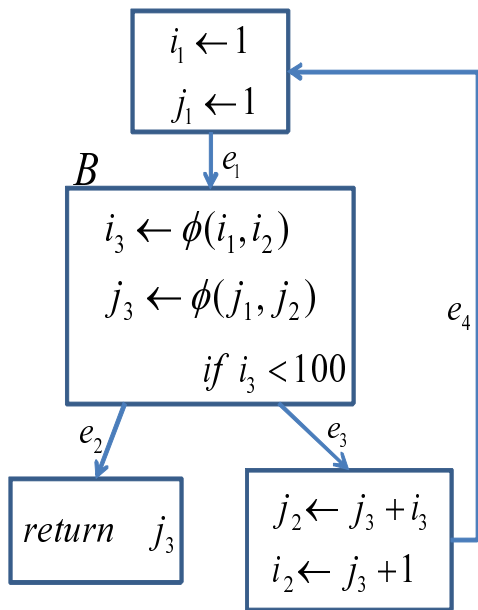


図 15: SSA 形式のプログラム P

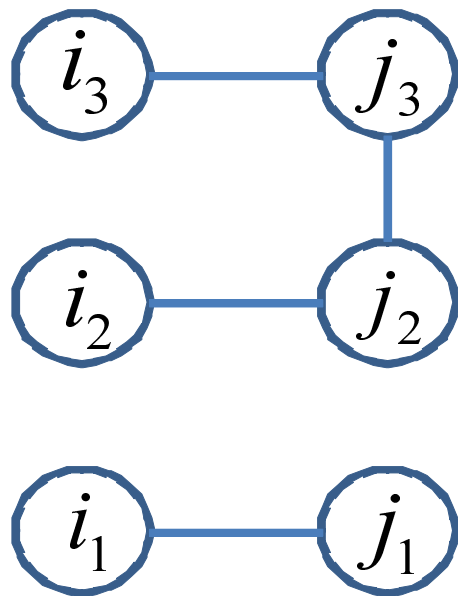


図 16: P の干渉グラフ

図 16 から最大クリークの大きさが 2 であるので、彩色数は 2 であることがわかる。このプログラムに対して伝統的な手法で ϕ 関数の除去を行っていく。ブロック中にある ϕ 関数を除去する際、コントロールフロー e_1 上に

$$\begin{array}{ll}
 i_3 & i_1 \\
 j_3 & j_3
 \end{array}$$

コントロールフロー e_4 上に

$$\begin{array}{ll}
 i_3 & i_2 \\
 j_3 & j_2
 \end{array}$$

を挿入する。ここで ϕ 関数除去後の干渉グラフである図 17 に注目してみる。

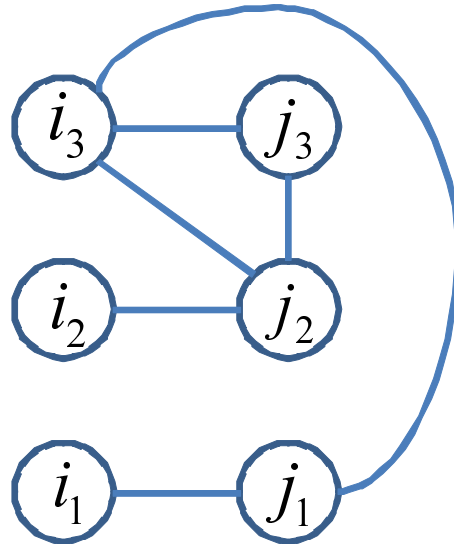


図 17: ϕ 関数除去後の干渉グラフ

ϕ 関数除去後の干渉グラフでは、 i_3 と j_1 、 i_3 と j_2 の干渉が増えているので、最大クリークの大きさが 3 になっている。 (i_3, j_2, j_3) によって、彩色数が 3 に増えてしまっている。以上のように、伝統的な手法を用いて ϕ 関数を除去する際、レジスタの要求を増やしてしまう場合がある。

レジスタの要求を増やすことは、再びスピルを行うというループの構造に戻ってしまう。そこでバルクコピーの特徴を考え直してみる。図 17 のレジスタ割当てを見てみると、フロー e_1 からブロック B に入った場合、レジスタ R_1 の値をレジスタ R_1 へ、レジスタ R_2 の値をレジスタ R_2 へ割当てて。すなわち ϕ 関数は何もしないということになる。しかし、フロー e_4 からブロック B へ入った場合、レジスタ R_2 の値をレジスタ R_1 へ、レジスタ R_1 の値をレジスタ R_2 へ一度に割当てて。すなわちレジスタ R_1 とレジスタ R_2 の値がレジスタ上で入れ替わっているのと同じということになる。

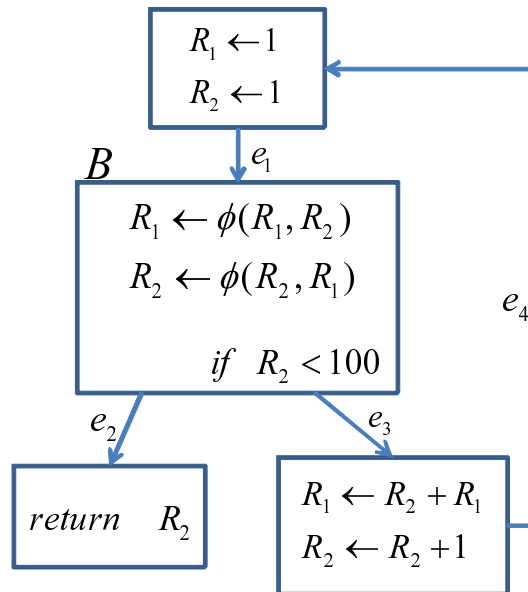


図 18: プログラム P のレジスタ割当て

このことを用いて、 ϕ 関数を以下のように扱っていく。

- × 一連のコピー
- レジスタ上の値の入れ替え (swap)

また、入れ替えがどのようになるか (効率よく入れ替えることができるか) は ϕ 関数の引数と結果がどのようにレジスタに割当てられるかに左右される。そこで、ここにおける合併を以下のタスクとして実装する。

合併 (coalesce) : 可能な限り多くの不動点 (fixed point) を持つようなレジスタ割当てを探す

不動点に関しては後の 7.3 で解説する。n の大きさの配列の入れ替えに関しては、レジスタの要求を増やすことなく一連の swap 命令で置き換えられる。その具体的な例を以下に示す。

label ℓ

| | |
|-------|---------------------------|
| R_2 | $\phi(\dots, R_1, \dots)$ |
| R_3 | $\phi(\dots, R_2, \dots)$ |
| R_1 | $\phi(\dots, R_3, \dots)$ |
| R_4 | $\phi(\dots, R_4, \dots)$ |

既存のものであれば、この ϕ 関数を一連のコピーとして扱う (図 19) のだが、ここでは swap 文を用いた入れ替えの動作 (図 20) として書き換えることができるのである。

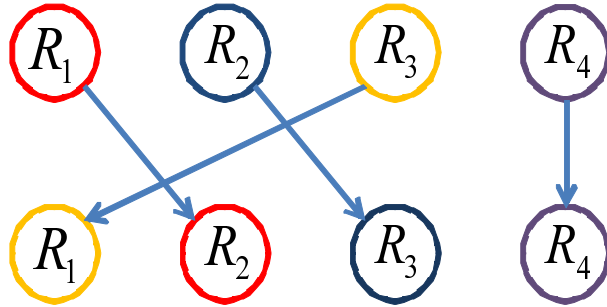


図 19: レジスタのコピー

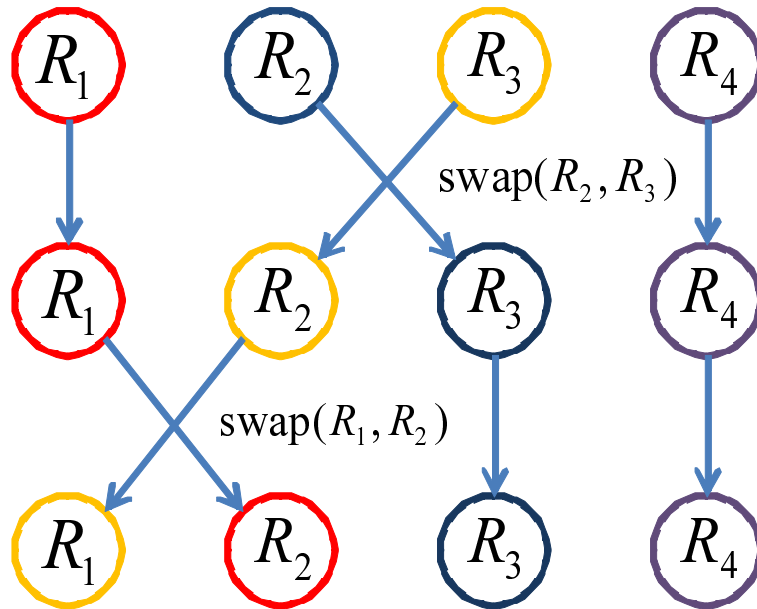


図 20: swap 命令を用いた ϕ 関数の除去

このようにして、SSA 逆変換を行う際、 ϕ 関数を `swap` 文に置き換えることにより、レジスタの要求を増やすことなく変換することができた。プロセッサが `swap` 命令を提供していない場合には、排他的論理和を用いることで同様な作業を行うことができる。またこの場合でも、レジスタの要求を増やすことはない。(n 個のレジスタの入れ替えであれば、その n 個のレジスタのみを用いて入れ替えることができる。)

例えば、レジスタ x とレジスタ y を入れ替える際、プロセッサが `swap` 命令を提供していなければ以下のように排他的論理和 \oplus を用いて実行すればよい。

$$\begin{array}{ll} x & x \oplus y \\ y & y \oplus x \\ x & x \oplus y \end{array}$$

もし ϕ 関数の場所においてレジスタに空きがあれば、そのレジスタをスペアとして利用し、入れ替えを行うことができる。空きがあるかについては、定理 5.3 と定義 6.2 からラベル ℓ において、いくつかのレジスタが使用されているかを容易に知ることができるため、空きがあるかどうかについても同様である。

しかし、逆変換を行う際に次のような場合コピー文は避けられない。

- ϕ 関数が同じ変数を複数回使用する場合
- ブロック内に生存区間をもつ変数を使用する場合

これらの場合、 ϕ 関数の引数、戻り値の変数の彩色結果が分かっているといけなため、関数のうちどれをコピー文として置き換えるかの決定は合併後に行う。

7.3 合併 (Coalescing)

ここまで見てきたように、 ϕ 関数はいくつかの場合を除きより多くのレジスタを必要とせずに消去することができた。すなわち、SSA 形式プログラムの干渉グラフの彩色は ϕ 関数を書き換えたレジスタ割当てと同値であることがわかる。ここで、SSA 形式の逆変換における ϕ 関数の消去の際に生じるレジスタの入れ替えの数を少なくするために、 ϕ 関数の不動点 (fixed point) の数を最大にする問題を導入する。

7.3.1 不動点 (fixed point)

coloring f : 彩色の結果を表す関数

ϕ 関数 : $y = \phi(\dots, x, \dots)$ に関して、 x と y が同じレジスタに割当てられているならば、すなわち、 $f(x) = f(y)$ (彩色された x と y の番号が同じであるということ) ならば、その x を ϕ 関数: $y = \phi(\dots, x, \dots)$ の不動点 (fixed point) という。

仮定から、この場合 ϕ 関数を消去する際にコード生成する必要がないことは明らかである。よって、不動点が多ければ多いほどそのコードは良いものとなる。ここでは、この不動点を可能な限り増やしていく。

7.3.2 コスト関数

不動点をいかにして増やしていくかの詳細なアルゴリズムに入る前に、タスクの中で用いていく関数を定義する。

SSA 形式プログラム P 、その干渉グラフを $G = (V, E)$ 、 P 中のすべての ϕ 関数の集合を Φ とおく。そして、有効な k 色のグラフ彩色を $f : V \rightarrow \{1, \dots, k\}$ としたときの、 ϕ 関数 $p : y = \phi(x_1, \dots, x_n)$ のコストを以下のように定義する。

$$C_f(p) = \sum_{i=1}^n \text{cost}_f(y, x_i) \quad \text{with } \text{cost}_f(a, b) = \begin{cases} w_{ab} & \text{if } f(a) \neq f(b) \\ 0 & \text{else} \end{cases}$$

そしてプログラム全体のコストは以下ようになる。

$$C_f(P) = \sum_{p \in \Phi} C_f(p)$$

今回の実装においては、 w_{ab} のそれぞれの値を一つ一つ求めず、固定値をもって評価した。すなわち、評価基準を 2 変数間のレジスタ交換が必要かということに絞った。

7.3.3 不動点を増やすためのアプローチ

既存のアプローチでは、干渉グラフのノードを合併することによってコードの最適化を行ってきた。そのための手法として、Hack らが [1] において提唱する A Heuristic Approach for SSA-Maximize-Fixed-Points を用いる。この手法の中で、 ϕ 関数の引数と、その結果を入れる変数に対して同色を割当てするために彩色しなおしていく。すなわちプログラム中の固定点を増やすために彩色しなおしていく。その際に 6.3.2 において導入したコスト関数に基づいて最適化を行う。既存の合併と違い、レジスタの要求を増やすこともなく、スピルしなおすこともない。

7.3.4 アルゴリズム

- 干渉グラフにおける ϕ 関数の結果と引数のノードを含むようなグラフ（これを相反グラフ (conflict graph) という）を同じ色に彩色することを試みる。

まず、それぞれの ϕ 関数に対して、最適化ユニット (optimization unit OU) を導入する。

$$OU : w = (y, x_1, x_2, \dots, x_m)$$

ここでの変数 x_i はすべて y と干渉しないもののみで構成される。 y と干渉する場合、そもそも y と同色に彩色することはできないからである。ここで、OU 中のそれぞれの変数

$(y, x_1, x_2, \dots, x_m)$ に対し、以下のようにして値を定義する。

$$x_i = w_{yx_i}$$

w_{yx_i} : ϕ 関数除去時、 x_i を y にコピーするために生じるコスト

OU におけるそれぞれの変数 (x_1, x_2, \dots, x_m) には、その変数を y にコピーするためにかかるコストが入る。 y に対しては、すべての OU に含まれている変数で、彩色しなおすことはないので、任意の値を代入しておく。それぞれの OU に対して別々に以下のコストの最小化アルゴリズムを適用していく。

[Init.]

y に対して許可されたそれぞれの色 c に対して $E_c = (c, C_c, S_c)$ を待ち行列に入れる。

c : c に対して許可された色

C_c : 相反グラフ (y, x_1, \dots, x_m を含むような干渉グラフのサブグラフを初期値とする)

S_c : C_c 中の最大の重みを持った組み合わせ (色 c に彩色することになっている相反グラフ中のすべてのノードを表す)

ここで E_c から得られるのは、 S_c に含まれるノードの重みの合計である。 S_c 中のノードは、現在の彩色において色 C に彩色予定のものなので、不動点となる変数を表している。つまり S_c 全体の重みが大きければ大きいほど ϕ 関数除去時のコストが少ないものとなる。待ち行列には、重みが降順になるように並べて入れていく。

[Test.]

待ち行列から E_c を一つ取り出し、彩色の改善を試みる。ここでは、テストフェーズが終了するまでは実際に彩色の変更をすることはない。それぞれの $u \in \{y, x_1, \dots, x_m\}$ に対して、彩色を c に変えていくが、もし u の隣接ノード n もまた c に彩色されていた場合、 u のもともと彩色されていた色を n に割当てる。

ノード u の彩色を変えた際に u の隣接ノード n と色の衝突が起きてしまった場合、 n の色 c を u の元の色に変えることによって解決する。もし n の色を変えらることによってさらに色の衝突が起きてしまった場合、再帰的にノードの色を変えていく。

ノード u を c に彩色することにより生じる n の色を入れ替える作業は、以下の3つのフェーズのどれかで終了する。

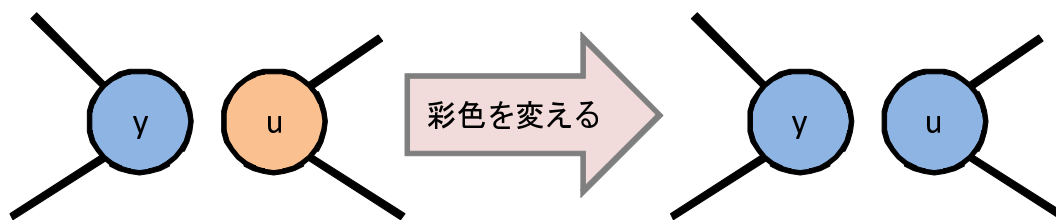


図 21: 彩色の変更

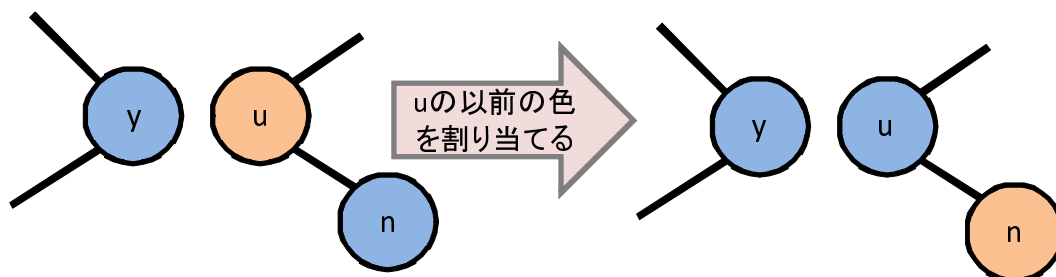


図 22: 色の衝突が起きた場合

- 1 n の色の入れ替えによって、新たな衝突が生じない
- 2 n の色は他の $OU(OU')$ によってすでにピンされている
- 3 現在の OU において n が pinning candidate である

2 の場合、 n の色を変えることは OU' でのコストを増加させる可能性があるので、辺 uu が相反グラフ C_c に加えられ、 u は C_c のすべての安定集合 S_c から除かれる。そして、 S_c は再計算され、待ち行列に再び入れる。

3 の場合、 u と n は互いに依存しあうことを意味し、このアルゴリズムは二つのノードを同じ色に彩色することはできない。ここで y が常に S_c に含まれるようにしたこと同様、もし $n = y$ ならば辺 uu か un を C_c に入れる。その後、 S_c を再計算し、待ち行列に再び入れる。

u の色を c に変えることによって生じるすべての色の衝突が解決したならば (すべてが 1 で終了するということ)、 u をピン候補としてマークする。そうならなければ、 u の色を変えることによって生じたすべての注釈を破棄する。

[Apply.]

テストフェーズにおいて、少なくとも 2 つの pin 候補がある場合 (y と 2 つ以上の x_i が同じ色に彩色することができるということ)、pin 候補から pinned に昇格させる。要するに、彩色方法をこれに決定するのである。そして、テストフェーズから得られたすべての彩色の

変更情報を G へ与える。

もしテストが失敗した場合それぞれのステップで辺が相反グラフに加えられるため、テストフェーズは必ず終了する。最悪の場合、安定集合には最終的に ϕ 関数の結果しか残らず、待ち行列に再び入れられない。そのため全体のアルゴリズムが終了するのである。

彩色の改善を行うことによって具体的にどのような利益が得られるかを以下の例を用いて述べる。

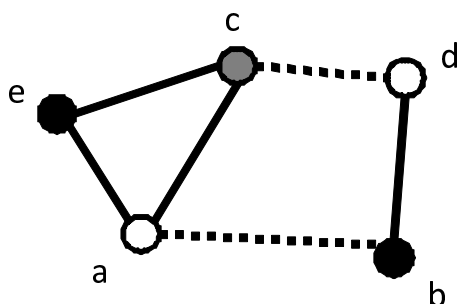


図 23: 合併前の干渉グラフ

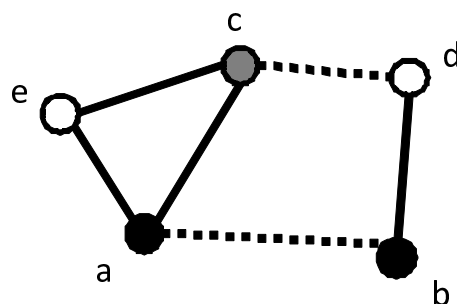


図 24: 合併後の干渉グラフ

図 23, 24 について、点線は ϕ 関数の引数、戻り値の組を表しており、干渉を表すものではない。すなわち、この点線上のノードの組をできる限り多く同色にすることがここでの目的である。まず、ノード a, b に注目する。このノードの組を同じ色にするために、a を b の色に彩色しなおす。これによって、ノード a とノード e において色の衝突が起きてしまうので、アルゴリズム通りノード e の色をノード a の以前の色に彩色しなおす。すると、図 24 のように a, b の組が同色となり、不動点を 1 つ増やすことができた。ノード c, d についてだが、これらを同色にするためには、すでに彩色しなおしたノード e の色を変えることになる。これはコストの増加になると考え、アルゴリズムでは彩色変更の中止をすることになっている。なのでノード c, d の彩色の変更は行わない。

以上のように彩色の変更（合併）を行った結果、合併前では 0 個であった不動点が合併後では 1 個に増えている。このようにして、プログラム全体で不動点を増やしていき、SSA 逆変換の際に必要なコピー分を最小限に抑えることが、合併の目的である。

8 実行結果の評価・考察

ここでは、実際にテストプログラムを用いて、実装を試みた SSA 形式を用いたレジスタ割当てにどのような効果があったかを調べた。

8.1 問題点

- レジスタ要求の増加

アルゴリズムの実装にあたって、最後のフェーズである SSA 逆変換の際に、レジスタの要求が増えてしまうという問題が生じた。レジスタ割当て全体をループのない構造にしているため、最終的な SSA 逆変換の際にレジスタの要求を増やしてはならない。もし増えてしまった場合、プログラム上の変数がレジスタに収まらない状況が出てくるので、再度スピルを行う必要がある。しかし、これではループの構造になってしまうので、SSA 形式のプログラムでのレジスタ割当ては行えても、ループを退避するという目的が達成されない。原因としては、SSA 逆変換の最後でやむ負えないコピー文の書き込みを行わなければならないところにある。この際に、レジスタの要求を増やしてしまう場合が出てくるのである。

8.2 解決方法

レジスタの要求が増えるという状況に備えて、余分に空のレジスタを用意しておく。もし、SSA 逆変換の際にレジスタの要求が増えたなら、用意しておいたレジスタを用いる。このことは、レジスタを有効に活用するという点では良くない方法であるが、SSA 形式を用いてループのない構造でレジスタ割当てをすることによるメリットの大きさを考え、この方法をとる。

8.3 実験

レジスタ要求の増加によるアルゴリズムの改善をしたため、結果が得られていない。今後プログラムの改善を引き続き行う。

9 おわりに

9.1 課題

今後の課題について述べる。

9.1.1 SSA 逆変換の問題

実験においても述べたように、SSA 形式を用いることによるメリットを最大限生かすために、スプイル後にレジスタ要求を増やすことは避けたい。今回の実装では、要求が増えた際に使用できるようなレジスタをあらかじめ確保しているが、これではレジスタの使用効率を上げるという点で良くない。レジスタの要求が増えるまでそのレジスタは空のままにしているからである。逆変換の際に起こるレジスタ要求の増加をなくすために、SSA 逆変換におけるアルゴリズムを改善すべきである。

9.1.2 コスト関数

今回の実装では、コスト関数に対して厳密に一つ一つのコストを定義していない。コピー文の変数を不動点とすることができなかつた場合、そのコピー文に対してコスト 1 と定義している。コピー文によっては、`swap` を用いるもの、コピー文をそのまま入れるものに分かれ、それによってコストは変わってくる。ここを厳密に定義し、合併を行うことができれば、よりよいコードに改善できる可能性がある。

9.2 関連研究

9.2.1 合併に関する研究

まず、グラフ彩色を用いたレジスタ割当てにおける合併の技術として、Chaitin[12] による積極的な合併 (aggressive coalescing) があげられる。コピー元とコピー先が干渉していない場合に合併するというものである。また、彩色数を増やすことのない合併の技術のために様々な研究がなされてきた。Briggs[13] による保守的な合併 (conservative coalescing)、George, Appel[14] による iterated coalescing などがある。

9.2.2 SSA 形式の干渉グラフに関する研究

SSA 形式における干渉グラフが弦グラフになることを Hack ら [15] は証明している。また、Brisk[16] は SSA 形式の干渉グラフの完全性に関する証明を与えている。SSA 形式におけるスプイル問題の難しさを Bouchez[17] が述べている。それぞれのラベルにおいて生存変数を k に減らすという問題をリロードを減らしつつ行うことは NP 完全であるという内容である。

参考文献

- [1] Sebastian Hack, Daniel Grund, and Gerhard Goos: Register Allocation for Programs in SSA-Form, 2007
- [2] COINS - Project. Coins - project home page. <http://www.coins-project.org/>.
- [3] COINS Project COINS Register Allocation
<http://www.coins-project.org/COINSdoc/backend/backend-frame.html>
- [4] COINS Project COINS バックエンド部の実装
<http://www.coins-project.org/COINSdoc/backend/backend-frame.html>
- [5] Bergner, P., Dahl, P., Engebretsen, D., O 'Keefe, M.: Spill code minimization via interference region spilling. In: PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation, New York, NY, USA, ACM Press (1997) 287-295
- [6] Chow, F.C., Hennessy, J.L.: The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.* 12 (1990) 501-536
- [7] Hsu, W.C., Fisher, C.N., Goodman, J.R.: On the Minimization of Loads/Stores in Local Register Allocation. *IEEE Trans. Softw. Eng.* 15 (1989) 1252-1260
- [8] Guo, J., Garzaran, M.J., Padua, D.: The Power of Belady 's Algorithm in Register Allocation for Long Basic Blocks. *The 16th International Workshop on Languages and Compilers for Parallel Computing (2003)*
- [9] Belady, L.: A Study of Replacement of Algorithms for a Virtual Storage Computer. *IBM Systems Journal* 5 (1966) 78-101
- [10] M.C.Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980
- [11] T.Lengauer, R.E. Tarjan, A fast algorithm for finding dominators in a flowgraph, *Trans. Programm. Languages Systems* 1 (1) (1979)121-141
- [12] Chaitin, G.J., Auslander, M.a., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via graph coloring. *Journal of Computer Languages* 6 (1981) 45-57
- [13] Briggs, P., Cooper, K.D., Torczon, L.: Improvements to graph coloring register allocation. *ACM Trans. Program.Lang.Syst.* 16 (1994) 428-455
- [14] George, L., Appel, A.W.: Iterated register coalescing. *ACM Trans. Program. Lang. Syst.* 18 (1996) 300-324
- [15] Hack, S: Interference Graphs of Programs in SSA-form. Technical Report 2005-15(2005)

- [16] Brisk, P., Dabiri, F., Macbeth, J., Sarrafzadeh, M.: Polynomial time graph coloring register allocation. In: In 14th International Workshop on Logic and Synthesis, ACM Press (2005)
- [17] Bouchez, F.: Allocation de registres et vidage en m ´emoire. Master 's thesis, ´ENS Lyon (2005)
- [18] 中田 育男. コンパイラの構成と最適化. 朝倉書店. 1999
- [19] Henry S. Warren, Jr. Hacker's Delight. 2003

謝辞

本研究を進めるにあたり多大なる御指導ご鞭撻を頂いた、東京工業大学数理・計算科学専攻教授の佐々政孝先生に深く感謝の意を表します。
また、佐々研究室の皆様にはさまざまな面で助力を頂きました。なんとってお礼を言えば良いか言葉も見つかりませんが、あらためまして、ここに深くお礼申し上げます。