

レジスタプロモーションによるコード最適化

東京工業大学
理学部
情報科学科

及川宗明
(0504350)

平成20年度卒業論文

指導教官 佐々 政孝 教授

2009年2月5日

目次

第1章	はじめに	3
1.1	背景	3
1.2	研究動機と目的	3
1.3	構成	3
第2章	COINS について	5
2.1	背景	5
2.2	構成	5
第3章	制御フローグラフ	7
3.1	制御フローグラフ	7
3.2	基本ブロック	8
3.3	有向辺	8
3.4	パス	8
3.5	先行ブロック	8
3.6	後続ブロック	8
3.7	自然ループ	8
3.8	サラウンディングループ	9
第4章	別名解析	10
4.1	別名解析について	10
4.2	flow-sensitive と flow-insensitive について	11
第5章	LIR	13
5.1	LIRの構成	13
5.1.1	L式の意味	13
第6章	レジスタプロモーション	15
6.1	レジスタプロモーションの概要	15
6.2	明確な参照、あいまいな参照について	15
6.3	アルゴリズム	16
第7章	レジスタプロモーションの設計と実装	19
7.1	レジスタプロモーション	19
7.1.1	目標	19
7.2	COINS 上のレジスタプロモーション	19
7.3	ポイントにおけるレジスタプロモーション	21

7.4	別名解析を用いたスピルアウト軽減	24
7.5	全体の構成	27
第 8 章	実行結果の評価と考察	28
8.1	実行環境	28
8.2	実行結果と考察	28
8.2.1	SPEC のベンチマークプログラムの結果	28
8.3	まとめ	29
8.4	今後の課題	30
8.4.1	手続き間への拡張	30
8.4.2	スピルアウトの軽減	30
第 9 章	関連研究	31
9.1	SSA 形式での促進	31
9.2	ALAT を用いた投機的レジスタプロモーション	31
第 10 章	まとめ	32

目 次

2.1	COINS 構成図	6
3.1	制御フローグラフの例	7
4.1	別名解析の例	11
4.2	flow-sensitive な別名解析の例	12
4.3	flow-insensitive な別名解析の例	12
5.1	LIR の内部構造	13
6.1	レジスタプロモーションの例	17
7.1	ポインタにおけるレジスタプロモーションの例	23
7.2	別名解析を用いたレジスタプロモーションの例	26
7.3	全体の構成	27
8.1	目的コードの実行時間	29

第1章 はじめに

1.1 背景

COINS[7]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に研究が進められているコンパイラ向け共通インフラストラクチャである。COINS[7]にはレジスタプロモーション最適化が備えられている。しかし、現在 COINS 上ではレジスタプロモーション最適化を行ってもまだメモリアクセスを必要とってしまう部分が存在する。

1.2 研究動機と目的

コンパイラは、一般的に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速いということが知られているので、レジスタ割り当てという処理を行い、プログラム中のスカラー変数を可能な限りレジスタに格上げする。しかし、種々の理由により、一部のスカラー変数はレジスタ割り当ての候補にすら入れることができないため、無限個のレジスタの存在を仮定する仮想レジスタを使ったとしてもどうしてもメモリアクセスを必要とする部分が存在する。

ここで、C 言語において素朴なレジスタ割り当てを仮定すると、次の二つはレジスタ割り当ての候補から外れる。

1. メモリアドレスを使用される変数
2. グローバル変数

両者ともアクセスされるメモリ上の位置が明確でないため、レジスタの値を維持できない場合があり、精密な解析をしない限り完全なレジスタ変数にはできない。このような変数を明確な範囲でレジスタに格上げする最適化がレジスタプロモーションである。

以前、COINS[7]上でのレジスタプロモーションの研究が行われている。しかし、限られた範囲でしかレジスタプロモーションを適用できていない。

本研究では、John Lu と Keith D. Cooper の論文「Register Promotion in C Programs」[2]で紹介されたアルゴリズムと、以前狩野さんが行った卒業研究 [4][5] を元に以前よりも拡張されたレジスタプロモーションを COINS 上に実装した。

1.3 構成

本論文の構成を以下に示す。第2章では今回実装を追加した COINS[7] についての説明を行う。第3章、第4章、第5章ではレジスタプロモーションを実装する上で必要となってくる制御フローグラフ、別名解析 [6]、LIR の説明を行う。第6章ではレジスタプロモ-

ションのアルゴリズム [2] の説明を行う。第 7 章では実装方法について、第 8 章では実験として実装したものを使用し、実際にレジスタプロモーションを適用した場合と、そうでない場合との比較を行い、第 9 章では関連研究を紹介する。

第2章 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(compiler infra structure, COINS) について説明を行う。

2.1 背景

COINS[7] 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きた。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。

2.2 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)、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。

本研究で用いるコンパイラ・インフラストラクチャCOINSの概念図を図2.1に示す。COINSでは、複数の入力言語、複数の目的機種に対応する2つの中間コードがある。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。

COINS のソースはすべて Java 言語で書かれている。

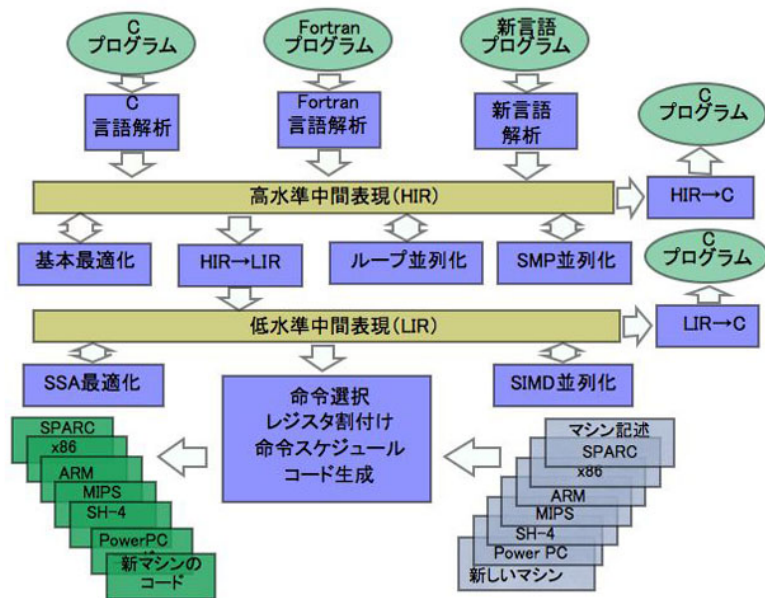


図 2.1: COINS 構成図

第3章 制御フローグラフ

本章では制御フローグラフについての説明を行う。

3.1 制御フローグラフ

プログラムの制御の流れをグラフで表したものを制御フローグラフと呼ぶ。制御フローグラフは、基本ブロックをノードとして、それらの間を分岐や合流を表す有向辺で結んだ有向グラフである。図 3.1 に制御フローグラフの例を示す。

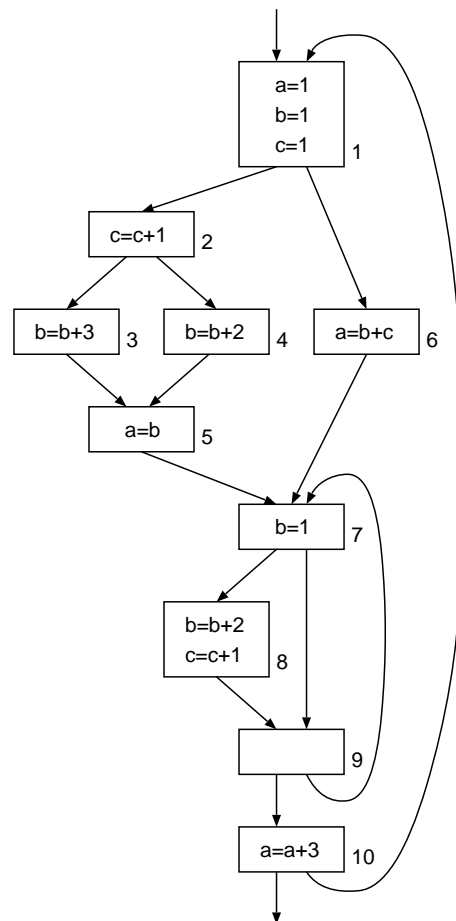


図 3.1: 制御フローグラフの例

3.2 基本ブロック

基本ブロック (*basic block*) とは、連続した文の列で、途中で飛越が起こらないものである。詳しくは次のように定義される。プログラムの最初の文、無条件あるいは条件飛び越しの行き先の文、無条件あるいは条件飛び越しの直後の文をリーダー (*leader*) という。リーダーから始まり次のリーダーの一つ前まであるいはプログラムの最後までの一連の文を基本ブロックという。以後これを単にブロックと呼ぶ。

3.3 有向辺

制御フローグラフにおいて、制御がブロック X からブロック Y に流れるとき $X \rightarrow Y$ と表記し、これを有向辺と呼ぶ。

3.4 パス

ノードの列 $X_0, X_1, X_2, \dots, X_i (i \geq 0)$ に対して、 $e_1 : X_0 \rightarrow X_1, e_2 : X_1 \rightarrow X_2, \dots, e_i : X_{i-1} \rightarrow X_i (i \geq 0)$ なる辺が存在すると仮定する。このとき X_0 から X_i にパス (*path*) が存在するという。

3.5 先行ブロック

$X \rightarrow Y$ であるとき、 X は Y の先行ブロック (*predecessor block*) とよび、 X の先行ブロックの集合を $pred(X)$ と表す。図 3.1 において $pred(5) = \{ 3, 4 \}$ である。

3.6 後続ブロック

$X \rightarrow Y$ であるとき、 Y は X の後続ブロック (*successor block*) とよび、 X の後続ブロックの集合を $succ(X)$ と表す。図 3.1 において $succ(7) = \{ 8, 9 \}$ である。

3.7 自然ループ

まず、自然ループを定義するにあたって必要な事項を説明する。

点

フローグラフの基本ブロック中の連続した文の間の地点を点 (*point*) という。基本ブロックの最初の文の直前と最後の文の直後も点という。

支配関係

フローグラフの初期頂点 (基本ブロック) から頂点 n_2 へ至るすべての路が必ず頂点 n_1 を通るとき、頂点 n_1 は頂点 n_2 を支配するといい「 $n_1 \text{ dom } n_2$ 」と記す。この定義では、すべての頂点は自分自身を支配することになる。図 3.1 において $7 \text{ dom } 9$ である。

帰辺

フローグラフ中の有向辺 $n_2 \rightarrow n_1$ は「 $n_1 \text{ dom } n_2$ 」が成り立つとき帰辺であるという。以上の準備のもとで、自然ループを定義する。

自然ループ

帰辺 $b \rightarrow h$ があるとき、この辺に関する自然ループ(*natural loop*)とは、 h と、 h を通らずに b に到達できるようなすべての頂点 (b を含む) をあわせたものである。自然ループは入り口点(*entry point*)となる頂点 (基本ブロック) がただ 1 つである。これをヘッダという。ヘッダはループ内のすべての頂点を支配する。ループを繰り返すための路、つまりヘッダへと戻る路 (帰辺) が少なくとも 1 つある。

二つの自然ループは、ヘッダが異なれば、互いに共通部分がないか一方のループが他方のループに完全に含まれることが知られている。ループ内にない後続ブロックをもつ (ループ内の) 基本ブロックを出口ブロックという。

なお、2 つの自然ループがあり、ヘッダを共有しているが互いに包含関係にないという場合がある。この場合は、この二つを合わせて一つの自然ループとして扱う。

3.8 サラウンディングループ

これは制御フローグラフとは直接関係ないものであるが、ここで説明しておく。ループ A とループ B があり、ループ A の要素ブロックが、すべてループ B の要素ブロックでもあるとき、つまりループ A がループ B に含まれるとき、ループ B をループ A のサラウンディングループ(*surrounding loop*) であるという。

第4章 別名解析

本章では以前、吉羽さんが研究した別名解析 [6] についての説明をする。

4.1 別名解析について

異なる変数や式が同じメモリアドレスに割り当てられるとき、これらは互いに別名であるという。別名が存在する可能性として

- ポインタ型
- 関数呼び出し

などがあげられる。

まず、ポインタ型について説明する。ポインタ p が変数 a を指している時、ポインタ p は変数 a と同じメモリアドレスを参照する可能性があるので別名であるといえる。

次に、グローバル変数を以下のようなプログラムを用いて説明する。

```
int globalVariable=0;

square() {
    globalVariable*=globalVariable;
}

main() {
    globalVariable=1;
    square();
}
```

main 関数では、グローバル変数である `globalVariable` を参照使用しており、`square` 関数においても同様である。`square` 関数を `main` 関数から呼び出しているが、`main` 関数の立場から見れば `globalVariable` の値は、直接 `globalVariable` のメモリ位置にアクセスすることなく `square` 関数を呼び出すだけで書き換えられている。つまり、`globalVariable` は `square` 関数に関して別名を持つ可能性があると考えられる。

別名解析 [6][16][17] とは、以上のように何と何が別名関係にあるのかを明らかにすることである。図 4.1 のようなプログラムに関して別名解析を行うと、結果は次のとおりになる。このフローグラフでは、ポインタ p は、 a または b を、ポインタ q は b を指すということになる。

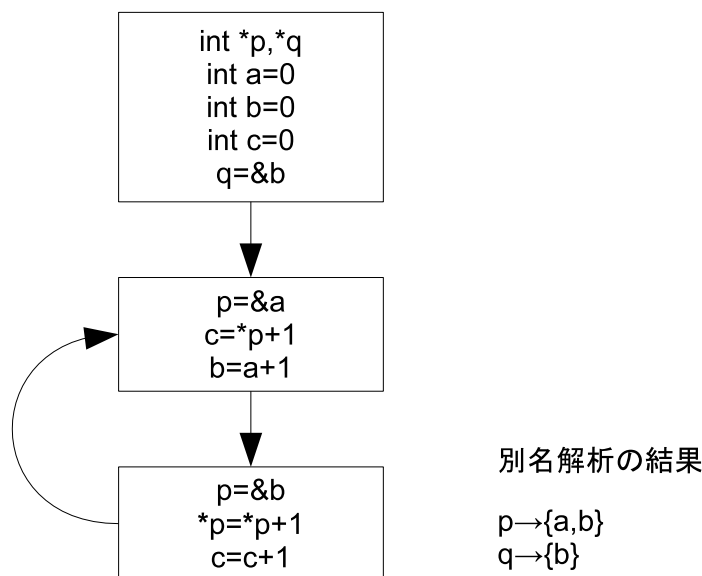


図 4.1: 別名解析の例

4.2 flow-sensitive と flow-insensitive について

別名解析には flow-sensitive なものと flow-insensitive なものが存在する (実は context-sensitive なものもある)。

flow-sensitive な解析はプログラムのフローを考慮した解析である。例えば、図 4.2 のような例が挙げられる。

このフローグラフには、パスの分岐があるが、それぞれのパスで別名関係がどのようになっているか、詳しい情報を与えてくれている。たとえば、ブロック B3 と B4 はプログラムの流れとしては分離している、つまり違うフローの中にある。そして、B3、B4 それぞれでポインタ *p が参照しているものも異なる。

flow-insensitive な解析は、プログラムのフローを考慮しない解析である。先ほどのフローグラフについて flow-insensitive な解析をしたものを図 4.3 に示す。

flow-insensitive な別名解析 [16][17] は分岐して別パスとなっているブロックの区別をせず、1つの手続き全体で別名関係の結果を出す。

ここで flow-sensitive な解析と、flow-insensitive な解析との違いを明確にする。

- *flow-sensitive* - フローの流れを考慮に入れた解析で、解析時間は長いですが、精度は高い
- *flow-insensitive* - 全ての文が実行されると仮定した解析で、解析時間は短いですが、精度は低い

本研究では flow-insensitive な別名解析 [6] を採用することにする。

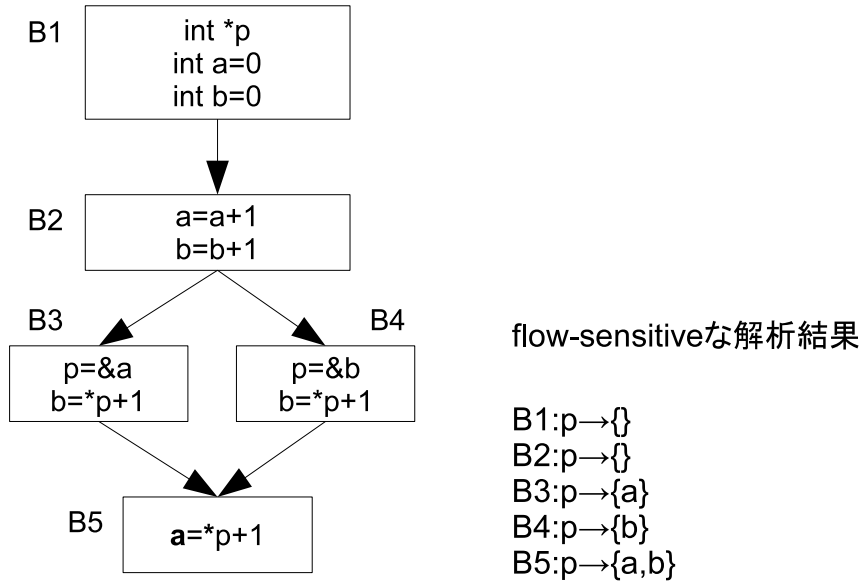


図 4.2: flow-sensitive な別名解析の例

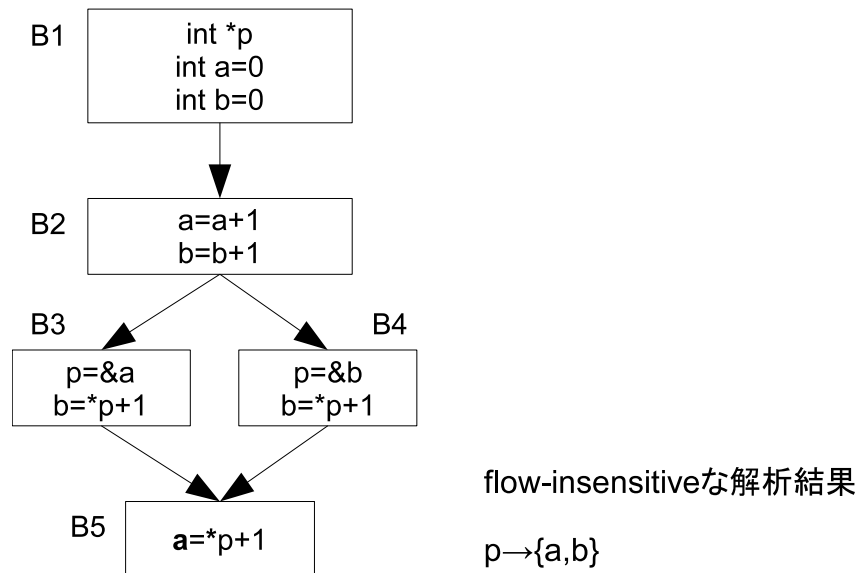


図 4.3: flow-insensitive な別名解析の例

第5章 LIR

レジスタプロモーションはLIR(低水準中間表現)を解析することによって実装する。よって、本章ではLIRの説明を行う。

5.1 LIRの構成

原始プログラムはL-module と呼ばれる LIR コードに変換される。L-module はグローバルシンボルテーブルといくつかのL-function を含む。L-function はローカルシンボルテーブルとL-sequence を含む。L-sequence はL式のリストであり、PROLOGUE式で始まりEPILOGUE式で終わる。図5.1にLIRの内部構造を示す。

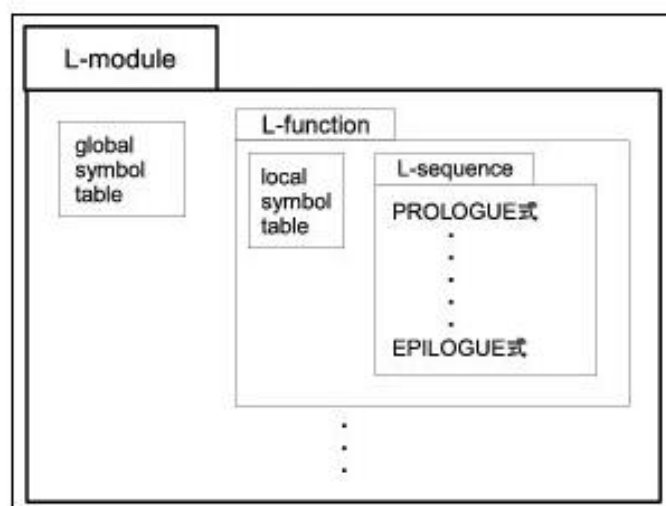


図 5.1: LIR の内部構造

5.1.1 L式の意味

ここで、一部のL式についてその意味を説明する。

Intconst 式 Intconst 式は整数を表す。(INTCONST I32 1) は 32 ビット整数の 1 を表す。

Floatconst 式 Floatconst 式は浮動小数点数を表す。(FLOATCONST I32 1.0) は 32 ビット浮動小数点数の 1.0 を表す。

Frame 式 Frame 式は、ローカル変数のメモリアドレスを表す。

Static 式 Static 式は、グローバル変数のメモリアドレスを表す。

Mem 式 Mem 式は、メモリ上の値を表す。たとえば、(MEM I32 (STATIC I32 "a")) はグローバル変数 a の値を表す。(MEM I32 (MEM I32 (FRAME I32 "p"))) はローカル変数 p が指し示す別名の値を参照する。

Set 式 Set 式は、第一引数の値に第二引数の値を代入する。

(SET I32 (MEM I32 (FRAME I32 "x")) (MEM I32 (FRAME I32 "y")))

は $x = y$ を表す。

Call 式 (CALL x_1 ($x_2 \dots x_n$) ($y_1 \dots y_m$)) は、 x_1 が表すメモリにある関数に引数 $x_2 \dots x_n$ を与え、結果の値を $y_1 \dots y_m$ に代入する。

これらの他にも命令は多数存在するが、本論文には書ききれないので詳しくは COINS のホームページ (COINS Project[7]) を参照してほしい。

第6章 レジスタプロモーション

6.1 レジスタプロモーションの概要

レジスタプロモーションとは、

1. メモリアドレスを使用される変数
2. グローバル変数

を明確な範囲でメモリアドレスからレジスタに格上げすることにより、コードの実行時間を改善することである。一般的にメモリへのアクセスよりレジスタへのアクセスのほうが速いことが知られているのでこの最適化はかなり効果的なものである。ループ文が最も実行時間に影響を及ぼすことが分かっているので、ループ文を対象として行われる。ループに入る前に明確であると判断された値はメモリアドレスからレジスタにロードする。ループ内では、この値の参照はレジスタへの参照に書き換えられる。ループを出るとき値はメモリアドレスにストアする。

6.2 明確な参照、あいまいな参照について

レジスタプロモーションを実装する際、促進できるものとできないものを見分けるために、明確な参照、あいまいな参照の区別が必要になるので以下に説明する。まず予備知識を説明する。

タグ

命令に使われうるメモリ位置の字面上の識別名を「タグ」という。例えばコードの中で `int a;` として整数型の変数 `a` を定義するとき、メモリ上にはこの変数 `a` の値をしまふ領域が確保される。この領域の字面上の識別名は `a` であり、これがタグである。つまり、`a` は単に変数名でもあり、またその値がしまわれているメモリ位置の識別名でもある。

さて、本題であるが、タグ `a` への直接参照を「`a` への明確な参照」という。例えば `a=a+1;` の `a` のように、`a` のメモリ位置を直接参照するものである。

タグ `a` への間接的な参照を「`a` へのあいまいな参照」という。`a` の別名が存在するとき、タグ `a` への `a` の別名からの参照は `a` へのあいまいな参照である。第4章で説明したようにポインタ、手続き呼び出しなどがあいまいな参照である。

コードのあるセクション A のあらゆる箇所で「a への明確な参照」があったとき、「セクション A で a は明確な参照を受ける」といい、セクション A の中に「a へのあいまいな参照」があったとき、「セクション A で a はあいまいな参照を受ける」という。

6.3 アルゴリズム

レジスタプロモーションは以下のように進めていく [2]。

1. ループ構造を見つける 手続き内においてループ構造 (自然ループ) を見つける。自然ループでないときはレジスタプロモーションの対象とはしない。ループ同士の包含関係も調べておく。
2. 初期情報を集める それぞれの基本ブロック b について二つの集合を計算する。 $B_Explicit_b$ は基本ブロック b 内の明確に参照されるすべてのタグの集合、 $B_Ambiguous_b$ は b においてあいまいな参照を受けるすべてのタグの集合である。
3. ループ内の解析 各々の自然ループ l について l に含まれる基本ブロックをたどり以下のものを求める。なお、 $surrounding-loop(l)$ とはループ l のサラウンディンググループのことである。

$$\begin{aligned}
 L_Explicit_l &= \bigcup_{b \in l} B_Explicit_b \\
 L_Ambiguous_l &= \bigcup_{b \in l} B_Ambiguous_b \\
 L_Promotable_l &= L_Explicit_l - L_Ambiguous_l \\
 L_Lift &= \begin{cases} L_Promotable_l & \text{if } l \text{ is an outermost loop} \\ L_Promotable_l - L_Promotable_{surrounding-loop(l)} & \text{otherwise} \end{cases}
 \end{aligned}$$

4. コードを書き直す 各ループについて $L_Promotable_l$ に含まれるタグそれぞれについて、仮想レジスタ v をつくる。さらにループ内におけるそれらのタグへの参照をすべて v への参照に変換する。
5. タグを促進する 仮想レジスタを使うように書き換えられたタグに関しては、促進可能な最外ループに入る直前に新しいブロックを挿入し、そこで仮想レジスタにロードする。また、そのループから出るときは、出た直後に新しいブロックを挿入し、そこでもとのメモリにストアする。

上に示した式について、少し説明を加えておく。

$L_Explicit_l$ 、 $L_Ambiguous_l$ はそれぞれループ内において明確な参照を受けているタグ、あいまいな参照を受けているタグの集合である。それぞれ、ループに含まれる基本ブロックの $B_Explicit_b$ や $B_Ambiguous_b$ の和集合を取っただけのものである。

$L_Promotable_l$ はループ l において促進できるタグの集合であり、各ループについて一度だけ計算される。

L_Lift_l はループ l において促進すべきタグの集合である。ループが二重以上になっている場合、同じタグであればできるだけ大きいループで促進するのが望ましい。小

さいループに関して促進したときよりも、より多くの値をレジスタへの参照に置き換えることができるからである。よって、ループ l を含むループがないとき、ループ l に関しては $L_Promotable_l$ の要素を促進すればよい。つまり $L_Lift_l = L_Promotable_l$ である。ループ l を含むループ m があり、どちらのループでも促進できるタグがあったとき、外側のループ m に関して促進したほうがよい。よってループ l に関して促進すべきタグの集合 L_Lift_l は、 $L_Promotable_l$ からループ l のサラウンディングループでも促進できるもの $L_Promotable_{surrounding-loop(l)}$ を除いた集合になる。

以上の内容を例を用いて示す。

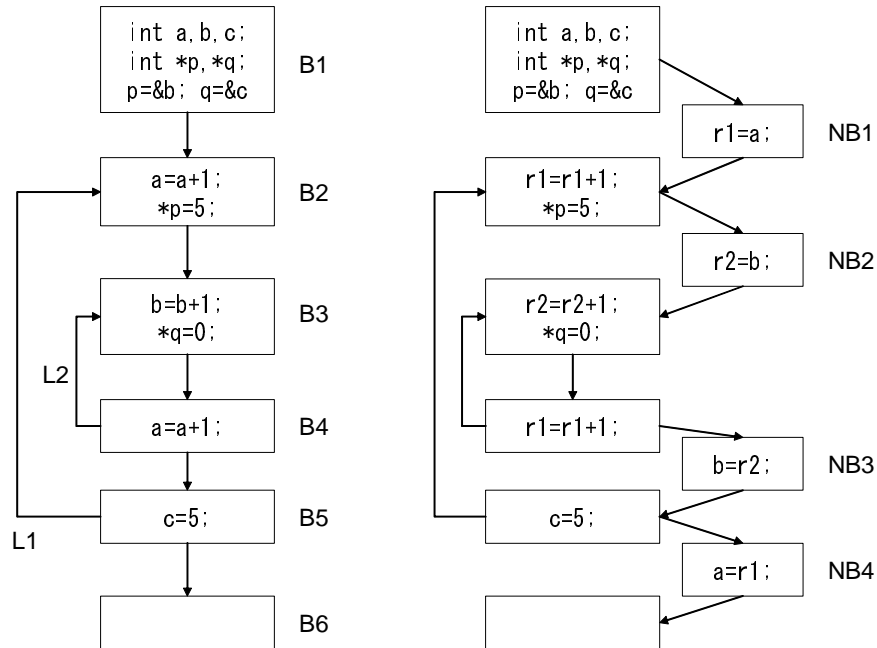


図 6.1: レジスタプロモーションの例

図 6.1 の 2 つの制御フローグラフは、左のものがプロモーション前、右のものがプロモーション後のものである。上の図でプロモーション前にも後にもある 6 つのブロックを上から B1, B2, B3...B6 とする。プロモーション後に加えられている 4 つのブロックを上から NB1, NB2, NB3, NB4 とする。外側のループを L1、内側のループを L2 とする。

まずは各ブロックについて調べる。上の例では間接参照は $*p$ と $*q$ のみなので、ブロックについての情報は下のようになる。

Block Information						
	B1	B2	B3	B4	B5	B6
B_Explicit	p,q	a	b	a	c	
B_Ambiguous		b	c			

これをもとに、次はループについての情報を調べる。結果は次の表のようになる。例えばループ L1 の $L_Explicit$ だが、ループの構成ブロックは B2, B3, B4, B5 であ

り、それぞれの B_Explicit の要素は、B2 で a、B3 で b、B4 で a、B5 で c なので、L1 の L_Explicit は a,b,c になる。同様にして 2 つのループに関して L_Explicit と L_Ambiguous を求める。L_Promotable は、単に L_Explicit の要素のうち L_Ambiguous に含まれないものを要素とすればよい。両方のループの L_Promotable が求まったら、最後に L_Lift を求める。L1 は最外ループなので、L1 の L_Lift は L1 の L_Promotable と同じものである。L1 は L2 のサラウンディングループなので、L2 の L_Lift は、L2 の L_Promotable から L1 の L_Promotable を除いたものになる。

Loop Information					
Loop	Blocks	L_Explicit	L_Ambiguous	L_Promotable	L_Lift
L1	B2-B5	a,b,c	b,c	a	a
L2	B3-B4	a,b	c	a,b	b

次にメモリからレジスタへのロード命令、レジスタからメモリへのストア命令を加える。ループ L1 について、入り口ブロック B2 に入る前に新しいブロック NB1 を、出口ブロック B5 を出た直後に新しいブロック NB4 を挿入する。ループ L1 で促進するタグは a なので、ブロック NB1 には a の値をメモリからレジスタ r1 にロードする命令「r1=a;」をセットする。ブロック NB4 には、レジスタからメモリに戻す命令「a=r1;」をセットする。ループ L2 についても同様の作業を行いレジスタプロモーションは終わる。

第7章 レジスタプロモーションの設計と実装

7.1 レジスタプロモーション

7.1.1 目標

COINS にはレジスタプロモーション最適化が既に組み込まれている [4]。しかし、レジスタプロモーションで対象となる変数はかなり強い制約を受けてしまっている。これでは、プログラムによってはあまり効果が見られないものも存在する。

COINS では、手続きごとに分割コンパイルを行うので、手続きの外部の情報を得ることができない。よって、全ての手続きで使用することができるグローバル変数については注意が必要である。本論文ではいくつかの段階に分けて、本研究で実装したレジスタプロモーションの説明を行う。

7.2 COINS 上のレジスタプロモーション

まず、狩野さんが行ったレジスタプロモーションについて説明する [4]。COINS の LIR においてグローバル変数のあいまいな参照が起こりうるのは次の場合である。

- (1) ループ l 内に、手続き呼び出しがある場合
- (2) ループ l 内に、ポインタなどによるメモリへの間接参照がある場合

(1) については、手続き呼び出し元のループ l 内のグローバル変数の値が呼び出し先で書き換えられうるので、ループ l 内で使われるグローバル変数はすべてあいまいな参照を受ける可能性がある。(2) は間接参照を受けている特定のタグがあいまいな参照を受けていることになる。ここで、 $L_Ambiguous_l$ を求めるために具体的にどのタグがあいまいな参照を受けているかを判定したい。COINS のバックエンドでは手続きごとにコンパイルが行われるので、そのとき最適化処理している手続きの外部の情報を知ることが難しい。よって(1)に関しては、ループ l 内で使われるすべてのグローバル変数のタグをあいまいな参照を受けるものとして扱う。

ここで、(1) を下記のような例を用いて説明する。

```

int a = 0;

square() {
    a *= a;
}

f() {
    int i = 0;
    for(; i < 10; i++) {
        a = a + i;
        square();
    }
}

```

ループにおいてレジスタプロモーションを試みるわけだが、ここで `a` は促進できない。なぜなら、`a` は手続き `square` で `a*=a` によって、手続き `f` の外で値が書き換えられているからである。ここで、もし `a` を促進してレジスタ `r` にしてしまうと、ループは次のように書き換えられる。

```

r = a;
for(; i < 10; i++) {
    r = r + i;
    square();
}
a = r;

```

手続き呼び出し先の `a*=a` により `a` の値は書き換えられている。しかし、`a` への直接参照を使っている `a=a+i` は `r=r+i` に書き換えられているため、促進先のレジスタ `r` に計算結果を保存する。よって、ループを出て、レジスタ `r` から `a` のメモリアドレスに値をストアし終わった時、`a*=a` の計算は無視されてしまい、結果的に `a` の値は `a=a+i` の計算のみ反映されてしまう。これより、手続き `f` のみを解析しただけではレジスタプロモーションできないことがわかる。

次に、(2) を下記のような例を用いて説明する。

```

int a = 0;
int *p = &a;

f() {
    int i = 0;
    for(; i < 10; i++) {
        a = a + i;
        *p = *p + i;
    }
}

```

この例でもループにおいてレジスタプロモーションを試みるわけだが、ここでも先ほどと同様に a は促進できない。なぜなら、 a は手続き f の外で $p=&a$ によってポインタ p にアドレスを取られていて、ループの中で $*p$ が使われているためである。ここで、もし a を促進してレジスタ r にしまうと、ループは次のように書き換えられる。

```
r = a;
for(; i < 10; i++) {
    r = r + i;
    *p = *p + i;
}
a = r;
```

$*p=*p+i$ はポインタ p の指す場所、つまり、 a のメモリ位置に $*p+i$ の結果を保存する式である。しかし、 a への直接参照を使っている $a=a+i$ は $r=r+i$ に書き換えられているため、促進先のレジスタ r に計算結果を保存する。よって、ループを出て、レジスタ r から a のメモリアドレスに値をストアし終わった後、 a の値は $a=a+i$ だけを計算し、 $*p=*p+i$ を無視したものになってしまう。

このようなことは避けなくてはならないが、 $p=&a$ は手続き f の外にあるため、手続き f の内部の解析だけではポインタ p が a を指していると判断できないことがわかる。

よって、ループ l において、手続き呼び出し、ポインタによるメモリへの参照があった場合、 l 内のすべてのタグはあいまいな参照をうけるものとして扱い $L_Ambiguous_l$ の要素に加える。

この方法の場合、促進できるものを促進しないことがある。つまり、考えうる最高のプロモーションに比べて精度は落ちるが、促進できないものを促進するということはないので、間違った最適化は避けられる。

7.3 ポインタにおけるレジスタプロモーション

本節では、ポインタにおけるレジスタプロモーションの手法を説明する。現在 COINS に組み込まれているレジスタプロモーションではポインタにおけるレジスタへの格上げは実装されていない。その理由としては、前節で説明したように、自然ループ内にポインタが存在した時点で、ポインタは他の変数と同じメモリアドレスの値を書き換えてしまう可能性があり、あいまいであると判断され、そのループ内の全ての変数がレジスタプロモーションの対象から外れてしまっていたからである。

しかし、C 言語ではポインタを用いた参照はよく使われるので、明確性が保たれた範囲でならレジスタプロモーションの対象としてポインタもレジスタに格上げできる。

ここで、下記のコードを用いて説明する。

```
int a = 0;
int *p = &a;
```

```

int b = 0;

f() {
    int i = 0;
    for(; i < 10; i++) {
        a = a + i;
        *p = *p + i;
        if(i == 5) {
            p = &b;
        }
    }
}

```

このコードの自然ループ内の変数の値を参照する際のL式を見てみると
変数 a は

```
(MEM I32 (STATIC I32 "a"))
```

変数 b は

```
(MEM I32 (STATIC I32 "b"))
```

ポインタ p は

```
(MEM I32 (MEM I32 (STATIC I32 "p")))
```

となっている。命令式 (MEM I32 (STATIC I32 "p")) はポインタ p が指す変数のメモリアドレス、つまり、この例では変数 a のメモリアドレスを取ってくるための命令である。この命令式をレジスタに格上げすることにすれば、ポインタ p が変数 a の別名であろうが、変数 b の別名であろうが明確性は保たれている。よって、ポインタ p は明確な参照である。そこでポインタ p をレジスタプロモートする。

ここで、(MEM I32 (MEM I32 (STATIC I32 "p"))) に対しレジスタプロモーションを適用した際のL式を見てみると

```
(MEM I32 (REG I32 "p% "))
```

となる。変数 a は上記のままである。

仮に (MEM I32 (MEM I32 (STATIC I32 "p"))) をレジスタに乗せたとする。この命令式はポインタ p が指す変数の値をとってくるための命令である。i=5の時、if文でポインタ p は変数 b の別名に書き換えられたにも関わらず、ポインタ p は変数 a の別名としてそのまま計算してしまうことになる。よって、(MEM I32 (MEM I32 (STATIC I32 "p"))) をレジスタに乗せることは不可能である。

ここで、一例を用いて説明することにする。

図 7.1 の左側の制御フローグラフはレジスタプロモーション適用前である。右側の制御フローグラフはレジスタプロモーション適用後である。まず、最初のループを見て

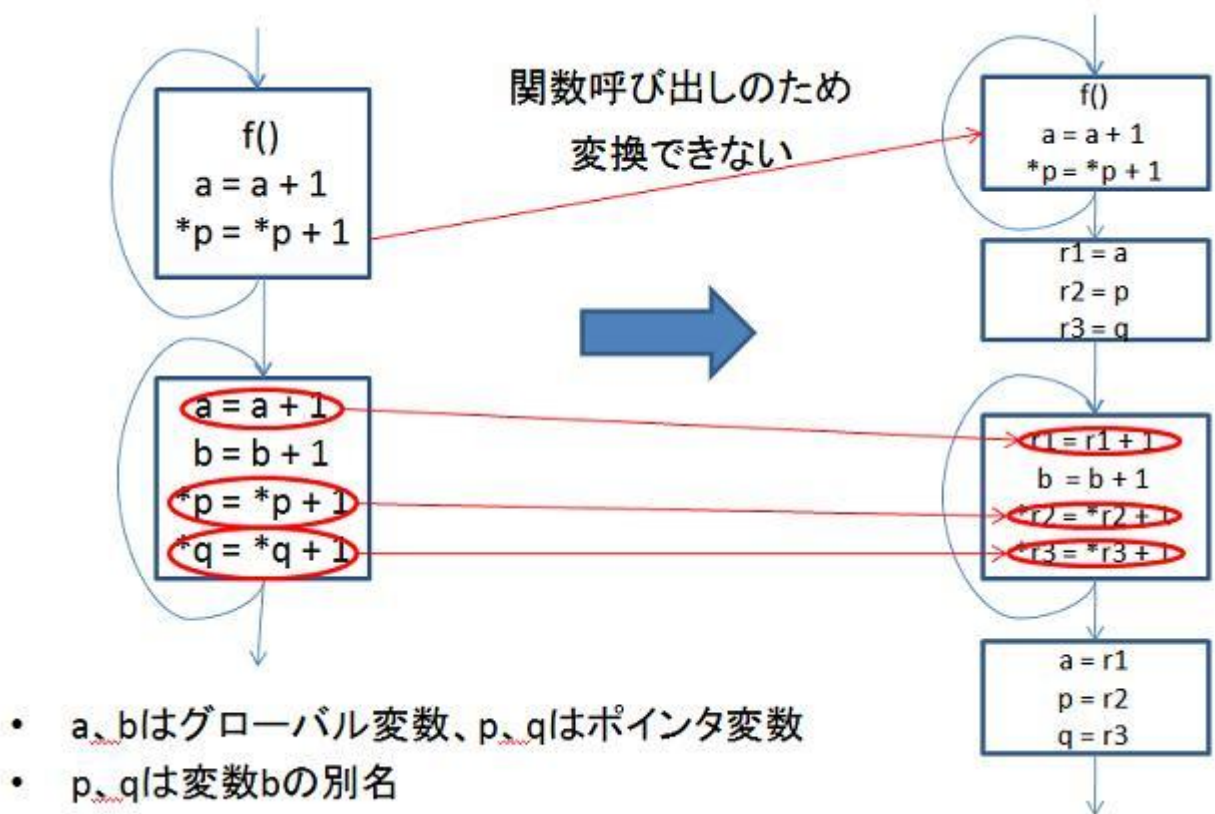


図 7.1: ポインタにおけるレジスタプロモーションの例

いただきたい。このループは関数呼び出し f が存在しているためにレジスタプロモーションは適用できない。次に、二つ目のループを見ていただきたい。このループは関数呼び出しが存在しないのでレジスタプロモーションを適用できることがわかる。よって、ループに入る直前でグローバル変数 a はレジスタ $r1$ 、ポインタ変数 p はレジスタ $r2$ 、ポインタ変数 q はレジスタ $r3$ に代入する。その後、ループ内のグローバル変数 a 、ポインタ変数 p 、 q をそれぞれのレジスタに変換する。最後に、ループを出た直後にレジスタの値をそれぞれの変数に代入する。

7.4 別名解析を用いたスピルアウト軽減

本節では別名解析 [6] を用いたスピルアウト軽減手法を説明する。

スピルアウトとは、レジスタに格上げした変数の個数が使用可能なレジスタ数を上回った場合、レジスタに格上げした変数をメモリへと一時的に退避しなければならないになってしまう。これをスピルアウトと呼ぶ。スピルアウトが起きてしまうと実行時間が遅くなることが知られているため、レジスタ合併 (register coalescing) などの最適化が行われる。また、プログラムが大きくなるほど必要となるレジスタ数が増えるのでスピルアウトが起こりやすくなってしまふ。よって、本手法では別名解析を用いることによって、同じメモリアドレスを参照する変数を同じレジスタに格上げすることにする。

たとえば下記のようなコードがあったとする。

```
int a = 0;
int *p = &a;
int *q = &a;
f() {
    int i = 0;
    for(; i <= 10; i++) {
        a = a + i;
        *p = *p + i;
        *q = *q + i;
    }
}
```

まず、このコードにおける自然ループ内の変数 p 、 q を参照する際の L 式を見てみる。

```
(MEM I32 (MEM I32 (STATIC I32 "p")))
(MEM I32 (MEM I32 (STATIC I32 "q")))
```

となる。別名解析の情報を用いないでレジスタプロモーションを適用すると、変数 p 、 q の L 式は前節で説明したようにそれぞれ

```
(MEM I32 (REG I32 "p% "))
(MEM I32 (REG I32 "q% "))
```

となる。ここで、別名解析の情報を使うと $p \rightarrow a$ $q \rightarrow a$ となる。これにより、変数 p 、 q は同じメモリアドレスを参照していることが分かるので同じレジスタに格上げすることが可能となる。よって、この2つは

```
(MEM I32 (REG I32 "p% "))
```

のようにでき、変数 p 、 q を参照するレジスタが上記のように一つで済むことがわかる。他の例でも説明しよう。

```
int a = 0;
int *p = &a;
int *q = &a;
int *r = &a;

f() {
    int i = 0;
    for(; i < 10; i++) {
        a = a + i;
        *p = *p + i;
        *q = *q + i;
        *r = *r + i;
    }
}
```

この例も同様に、別名解析の情報を使うと $p \rightarrow a$ $q \rightarrow a$ $r \rightarrow a$ となるので、変数 p 、 q 、 r は同じメモリアドレスを参照しているので、同じレジスタに格上げすることができる。

しかし、あるポインタが二つ以上のメモリアドレスを参照する可能性がある場合は必ずしも同じ変数を指すとは限らないので、異なるレジスタを使わざるを得なくなってしまう。これも例を用いて示す。

```
int a = 0;
int *p = &a;
int *q = &a;

f() {
    int b = 0;
    q = &b;
    int i = 0;
    for(; i < 10; i++) {
        *p = *p + i;
        *q = *q + i;
    }
}
```

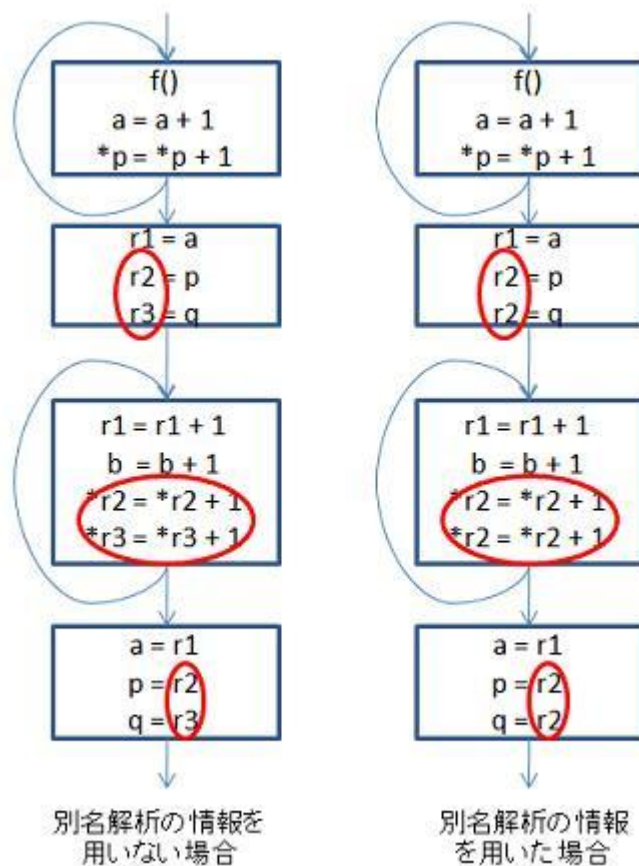


図 7.2: 別名解析を用いたレジスタプロモーションの例

この例では、別名解析の情報を使うと $p \rightarrow a$ 、 $q \rightarrow a, b$ となる。flow-insensitive な解析なので、ループの中でポインタ p が変数 a を指すか、変数 b を指すかわからないため、必ずしもポインタ p とポインタ q が同じメモリアドレスを参照するかどうか判断できない。よって、同じレジスタを使うことができない。

以上のように、別名解析を用いることによって、多少でもスピルアウトの発生を防ぐことができる。

ここでも、一例を用いて説明することにする。

図 7.2 は図 7.1 を別名解析の情報を用いてさらに変換したものである。図 7.2 の左側の制御フローグラフは図 7.1 の右側の例、つまりポインタにおけるレジスタプロモーションの適用後である。右側の制御フローグラフは別名解析の情報を用いてさらに変換したものである。ここで、図 7.1 の条件を見てみると $p \rightarrow b$ 、 $q \rightarrow b$ となっている。よって、ポインタ変数 p 、 q は同じメモリアドレスを参照しているのと同じレジスタに格上げ可能であることがわかる。図の丸で囲んだ部分に変更部分である。別名解析の情報を用いない場合レジスタが $r1$ 、 $r2$ 、 $r3$ と三つ使っているが、別名解析の情報を用いた場合レジスタが $r1$ 、 $r2$ と二つの使用で済んでいる。もし、ループ内で多くのレジスタを使用している場合、この変換はスピルアウトの軽減につながる。

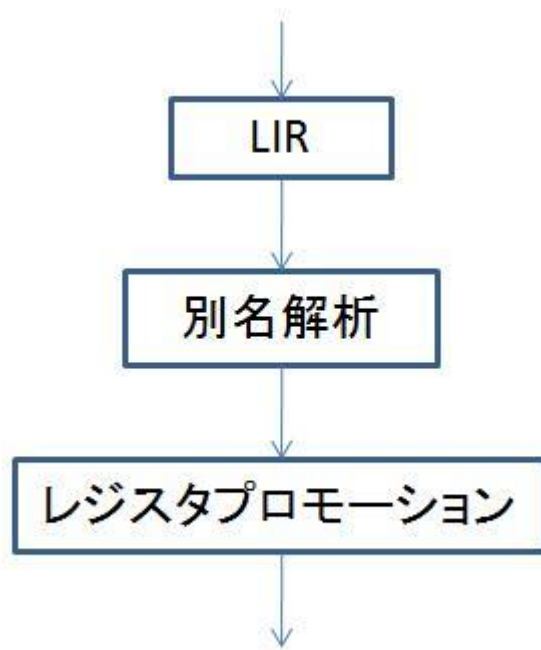


図 7.3: 全体の構成

7.5 全体の構成

本研究の構成を図 7.3 に示す。

図 7.3 のようにまず、LIR(低水準中間表現)を入力とする。LIR を解析することで別名解析を行う。次に、別名解析の情報をもとにポインタにおけるレジスタプロモーションを実装する。実装方法は本章で示した通りである。

第8章 実行結果の評価と考察

この章では実際にテストプログラムを用いて、レジスタプロモーションを適用した場合と、適用しなかった場合とで比較を行った。

8.1 実行環境

実験は、Sun Microsystems の、Sun Blade 1000 で行った。この主な仕様は、

アーキテクチャ	Superscalar SPARC Version 9
プロセッサ種別	750MHz UltraSPARCIII
1次キャッシュ	64KB データ、32KB インストラクション
2次キャッシュ	8MB 外部キャッシュ
メモリ容量	1GByte
オペレーティング環境	SunOS 5.8

表 8.1: Sun Blade 1000 の主な仕様

テストプログラムには以下のものを用意した。

SPEC CPU2000 のベンチマークプログラムより以下の9つ。

164.gzip, 181.mcf, 197.parser, 254.gap, 255.vortex, 256.bzip2, 300.twolf,
179.art, 183.quake, 188.ammf

SPEC のプログラムをそれぞれ3回ずつ実行し、実行時間の平均を取った。なお COINS のプログラムの実行時間計測には、time コマンドを用いた。なお、time コマンドでは1.2% の誤差は意味を持たない。また、coins-1.4.4.1 を用いた。

8.2 実行結果と考察

8.2.1 SPEC のベンチマークプログラムの結果

レジスタプロモーションを適用しない場合、狩野さんが行ったレジスタプロモーション [4]、本研究で行ったレジスタプロモーションを適用した場合を比較した結果は、図 8.1 のようになった。

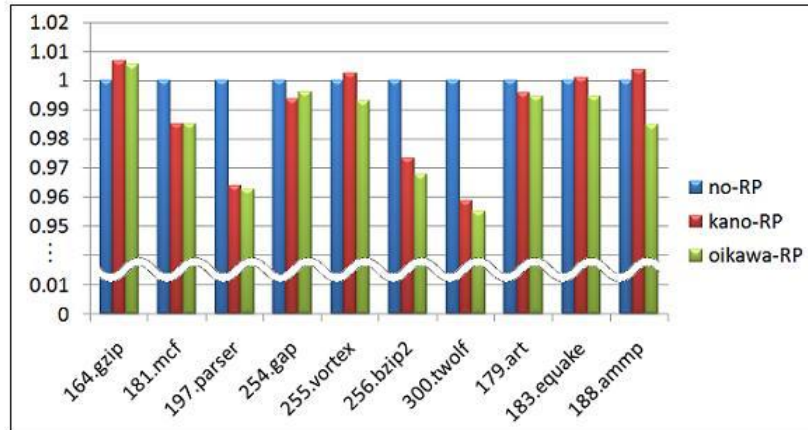


図 8.1: 目的コードの実行時間

図 8.1 を見てわかるように全体的に速くなったことがわかる。特に 300.twolf では約 4.5 % も実行時間の改善がみられる。また、188.ammp では、今回のポイントにおけるレジスタプロモーションを行ったことによる効果が最も顕著に表れた例である。164.gzip では適用できる変数も少なく、レジスタ使用数が増えたためのスピルアウトによる影響で実行時間が遅くなってしまった。また、レジスタプロモーションが適用できた変数の個数は狩野さんの行ったレジスタプロモーションよりも確実に増えていた。多いもので 40 個もの変数がレジスタに格上げされている。しかし、レジスタに多く格上げされているからと言って、必ずしも大幅な実行時間の改善は望めない。なぜなら、実行時間に一番影響するのはレジスタに格上げされた変数の個数よりも、命令が何度実行されたかに強く依存しており、また、スピルアウトにより実行時間が遅くなってしまうこともある。本研究のレジスタプロモーションでは別名解析 [6] を用いた時の効果はあまりなかった。狩野さんの行ったレジスタプロモーションの制約を緩める、つまり、ポイントにおけるレジスタプロモーションによる効果が本研究で示された。

8.3 まとめ

ポイントの扱が多い C 言語では、本研究で行ったレジスタプロモーションを適用するとかかなりの効果がみられる。しかし、プログラムによっては、レジスタ変数があまりに多すぎるためにスピルアウトなどが起こってしまい、実行時間が遅くなってしまいう場合もあるようだ。flow-sensitive な別名解析を行うことにより、さらに厳密なレジスタプロモーションを行うことができる。

8.4 今後の課題

8.4.1 手続き間への拡張

本研究ではループ内に関数呼び出しがあった場合は、COINS が分割コンパイルするためレジスタプロモーションを適用できなかった。しかし、ループ内での関数呼び出しは少なくないため、改善したい項目の一つである。

8.4.2 スピルアウトの軽減

レジスタに格上げする変数が多すぎるとスピルアウトが起りやすくなり、実行時間が遅くなってしまう可能性がある。これを回避するため促進する変数を制限する必要がある。

第9章 関連研究

レジスタプロモーションに関連する研究を紹介する。

9.1 SSA形式での促進

A.V.S Sastry[13]らの論文 [A new algorithm for scalar register promotion based on SSA form] では、SSA形式を用いることで、ロードやストアを頻繁に実行されるパスから、あまり実行されないパスに移すことによる実行時間改善の手法が提案されている。

9.2 ALATを用いた投機的レジスタプロモーション

Jin Lin[15]らの論文 [Speculative register promotion using Advanced Load Address Table (ALAT)] では、ALAT(Advanced Load Address Table)を用いた、投機的なレジスタプロモーションを行い、部分的な冗長性を省くことによる実行時間改善の手法が提案されている。最大7%もの実行時間改善がみられる。

第10章 まとめ

C言語はポインタを扱う処理が多いため、本研究のポインタにおけるレジスタプロモーションはかなりの効果がみられた。しかし、レジスタに格上げする変数が多いと、スピルアウトが起こってしまい実行時間が遅くなってしまう。よって、実行される変数が少ないものを解析することによりレジスタへの促進を制限すれば、更なる改善が考えられる。

また、本研究では手続き間のレジスタプロモーションを実装できなかった。もし手続き間で適用できるなら、かなりの効果がみられるだろう。

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、卒業生の狩野先輩、吉羽先輩にはさまざまな面で助力を頂きました。あらためまして、ここに深くお礼申し上げます。

参考文献

- [1] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, Peng Tul. Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores, 1998.
- [2] Keith D. Cooper and John Lu. Register Promotion in C Programs, 1997
- [3] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew. Speculative Register Promotion Using Advanced Load Address Table (ALAT), 2003
- [4] 狩野祐介. グローバル変数のレジスタプロモーションの実装, 東京工業大学理学部情報科学科卒業全文, 2005. <http://www.is.titech.ac.jp/sassa/lab/papers-written/kanon-thesis.pdf>.
- [5] 狩野祐介, 佐々政孝. 素朴なレジスタプロモーションの実装と評価, 日本ソフトウェア科学会大会論文集, 第 22 回, 5B-3(2005 年 9 月).
- [6] 吉羽和之. 型推論に基づく手続き間ポインタ解析アルゴリズムの実装, 東京工業大学理学部情報科学科卒業全文, 2006. <http://www.is.titech.ac.jp/sassa/lab/papers-written/9927000.pdf>.
- [7] COINS Project. Coins homepage. <http://www.coins-project.org/>.
- [8] 佐々 政孝. プログラミング言語処理系. 岩波書店, 1989.
- [9] 中田 育男. コンパイラの構成と最適化. 朝倉書店, 1999.
- [10] Matthew Postiff, David Greene, Trevor Mudge. The Store-Load Address Table and Speculative Register Promotion. 2003
- [11] COINS Project. COINS プロジェクト LIR 仕様書, 2002. <http://www.coins-project.org/spec/lir.pdf>.
- [12] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, Peng Tul. Register promotion by sparse partial redundancy elimination of loads and stores, 1998.
- [13] A.V.S. Sastry, Roy D.C. Ju. A new algorithm for scalar register promotion based on SSA form, 1998.
- [14] Patrick Carribault, Albert Cohen. Applications of storage mapping optimization to register promotion, 2004.

- [15] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew. Speculative register promotion using Advanced Load Address Table (ALAT). 2003.
- [16] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 32-41, 1996.
- [17] Andersen, L. O. Program Analysis and Specialization for the C Programming Language, PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, May 1994.