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

狩野　祐介
(0107011)

平成17年卒業論文

指導教官 佐々 政孝 教授

2月 6日

目 次

第 1 章 はじめに **3**

1.1 背景 3

1.2 概要 3

第 2 章 制御フローグラフ **5**

2.1 制御フローグラフ 5

2.2 基本ブロック 6

2.3 有向辺 6

2.4 パス 6

2.5 先行ブロック 6

2.6 後続ブロック 6

2.7 自然ループ 6

2.8 サラウンディングループ 7

第 3 章 レジスタプロモーション **8**

3.1 レジスタプロモーションの概要 8

3.2 明確な参照、あいまいな参照について 8

3.3 アルゴリズム 9

第 4 章 実装の準備、設定 **12**

4.1 COINS について 12

4.1.1 背景 12

4.1.2 構成 12

4.2 グローバル変数のレジスタプロモーション 12

4.2.1 目標 12

4.2.2 レジスタプロモーションの精度 13

第 5 章 設計と実装 **15**

5.1 はじめに 15

5.2 COINS のバックエンドについて 15

5.3 全体構造 16

5.4 各段階の説明 17

5.4.1 RPloop のインスタンスの生成 17

5.4.2 促進可能なタグの計算 18

5.4.3 命令の書き換えと追加 19

第 6 章 実行結果の評価と考察 **21**

6.1 実行環境とテストプログラム 21

6.2 実行結果と考察 21

6.2.1 COINS の開発プログラムの結果 21

6.2.2 SPEC のベンチマークプログラムの結果 23
6.2.3 まとめ 24
6.3 今後の課題 24
6.3.1 手続き間への拡張 24
6.3.2 ポインタ解析を行う促進 24
6.3.3 促進個数の制限 25

第 7 章 関連研究 26
7.1 手続き間への拡張 26
7.2 部分冗長除去の利用 26
7.3 SSA 形式での促進 26

第 8 章 おわりに 27

第1章　はじめに

1.1　背景

近年プロセッサの持つ並列処理能力が向上しているが、ALU や FPU などとは異なりロード・ストアユニットの処理並列度はメモリアクセス命令間の依存の解析が難しいため向上させるのが困難である。また、今後プロセッサの並列度が上がるにつれこの傾向は顕著になると考えられる。一方で、プログラム中にはメモリアクセス命令が少なからず含まれており、プロセッサ処理のボトルネックになると考えられる。この問題に対処するため、ハードウェアだけでなく、コンパイラの分野でもキャッシュの活用、レイテンシの隠蔽などのさまざまな最適化の研究がなされてきた。この中で、メモリアクセス回数を減らすレジスタプロモーション (Register Promotion) という最適化も注目されている。

1.2　概要

現代の RISC 型計算機のコンパイラは通常レジスタ割り当てという処理を行って、プログラム中のスカラー（非配列）変数を可能な限りレジスタ変数にする。しかし、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っていないため、仮にレジスタが無限に使えたとしてもメモリアクセスが少なからず残る。最近メモリアクセスを減らすために論理レジスタを６４あるいは１２８個備えたようなプロセッサも現れているが、これでは拡張したレジスタも有効利用できない。

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てる。

1. メモリアドレスを使用される変数
2. グローバル変数

の２種類であり、両者ともアクセスされる位置が明確でないために、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。例えば、複数の名前が一つの値のために存在したら、それは定義するごとにメモリーに保存しなくてはならないし、それぞれ使う前にメモリーからロードしなくてはならない。ポインターの存在がこの問題を表している：ポインタに参照されうる変数のセットについて、具体的な情報がなければ、保守的方法では、ポインタが指しうるすべてのメモリ領域を扱わなくてはならない。しかし、これらの変数もアクセス位置が明確な範囲ではレジスタ変数に格上げすることができる。この最適化がレジスタプロモーションである。

本研究では、John Lu と Keith D. Cooper の論文「Register Promotion in C Programs」で紹介されたアルゴリズムをもとに、レジスタプロモーションを実装する。

本論文の構成は以下の通りである。

第２章ではレジスタプロモーションとその実装を説明するために必要な、制御フローグラフに関する予備知識の説明を行う。第３章ではレジスタプロモーションに関係する用語の説明をし、その後「Register Promotion in C Programs」[Lu and Cooper 1997] で紹介されたアルゴリズムを例を使って説明する。第４章では、実装にあたっての前提条件とその説明、実装する環境の説明を行う。第５

章では、実装環境に関して、実装に必要な知識の説明と、実装したコードの内部構造やアルゴリズムの説明を行う。第 6 章では、実装したものを使って実際にレジスタプロモーションを行い、その結果を比較、評価する。

第2章　制御フローグラフ

レジスタプロモーションの実装にあたって、また本論文の説明にあたって制御フローグラフの知識が必要となる。本章では制御フローグラフに関する説明を行う。

2.1　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 2.1 に制御フローグラフの例を示す。

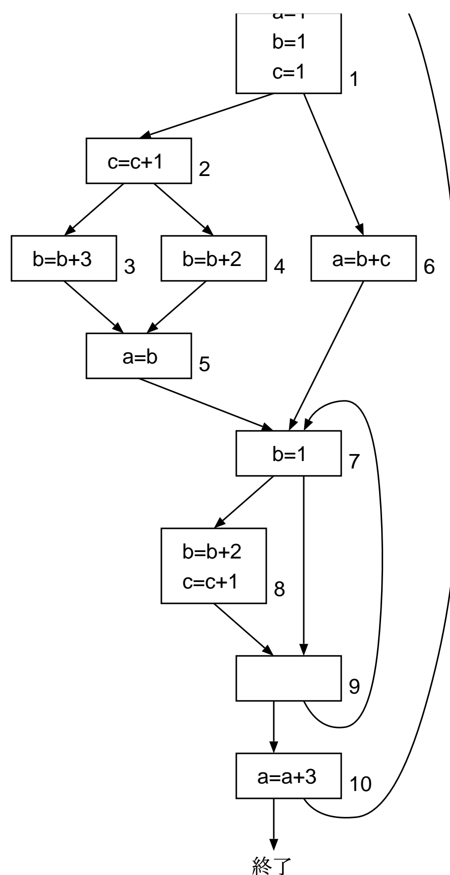

図 2.1: 制御フローグラフの例

2.2 基本ブロック

基本ブロック(*basic block*) とは、連続した文の列で、途中に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。(手続き呼び出し文は通常それ一つで基本ブロックとみなす)。以後これを単に**ブロック**と呼ぶ。

2.3 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 $X \to Y$ と表記し、これを**有向辺**呼ぶ。

2.4 パス

ノードの列 $X_0, X_1, X_2, \ldots, X_i\,(i \geq 0)$ に対して，$e_1 : X_0 \to X_1, e_2 : X_1 \to X_2, \ldots, e_i : X_{i-1} \to X_i\,(i \geq 0)$ 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．

2.5 先行ブロック

$X \to Y$ 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 2.1 において $pred(5) = \{3, 4\}$ である。

2.6 後続ブロック

$X \to Y$ 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 2.1 において $succ(7) = \{8, 9\}$ である。

2.7 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 *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 2.1 において 7 *dom* 9 である。

帰辺

フローグラフ中の有向辺 $n_2 \to n_1$ は「n_1 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 $b \to h$ 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。

2.8 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3章　レジスタプロモーション

3.1　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 (レジスタ促進) とは、プログラムのコードのいくらかの部分で、普通はメモリに入っている値をレジスタに移して、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もレジスタへのアクセスのほうが早いのでこの最適化はかなり効果的なものである。これは値をメモリからレジスタに移しても意味が変わらないようなコードのセクションに対して行われる。そのようなセクションに入る前に値はメモリからレジスタに移される。セクション内では、この値の参照はレジスタへの参照に書き換えられる。セクションを出るとき値はメモリに戻される。

第１章でも述べたが、本論文であつかう方法は Keith D. cooper らの「Rejister Promotion in C Programs」[Lu and Cooper 1997] で提案したものである。

3.2　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` の値をしまう領域が確保される。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。

別名

二つ以上の変数や式が同じメモリ番地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。ポインタ型などがあると、異なった変数が同じメモリ位置を指すことがあるので、別名の可能性がある。

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。例えば、ポインタ `p` が `a` のメモリ位置を参照しているとき、`a = *p + 1;` の `*p` のように、`a` のメモリ位置を間接参照するものは `a` へのあいまいな参照である。また手続き呼び出しもあいまいな参照となりうる。例えば手続き `f` が呼ばれたとき、`f` の中でタグ `a` を参照するような命令があった場合、これは `a` へのあいまいな参照である。なぜなら `f` を呼び出すだけで自動的に `a` のメモリ位置に対する参照が行われてしまうからである。

コードのあるセクション A の中に「`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。以下に例を示す。

```
int main(){
  int i;
  int *p;
  sum = 0;
  p = &sum;
  //ループ1
  for (i = 1; i <= 100; i++) {
       sum = sum + 1;  //イ
       *p = sum + 1;  //ロ
  }
  //ループ2
  for(i=1; i <= 10; i++){
     sum = sum + 5;  //ハ
  }
}
```

この例では、p = &sum により、ポインタ p が sum のアドレスを参照しているので、*p と sum は互いに別名である。イで sum は明確な参照である。ロで*p は sum へのあいまいな参照である。ハで sum は明確な参照である。これらをまとめるとループ 1 において sum はあいまいな参照と明確な参照を受けており、ループ 2 においては、*p が使われていないので sum は明確な参照だけを受けている。

3.3 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。

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。$\texttt{B_Explicit}_b$ 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、$\texttt{B_Ambiguous}_b$ 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ループのことである。

$$\texttt{L_Explicit}_l = \bigcup_{b \in l} \texttt{B_Explicit}_b$$

$$\texttt{L_Ambiguous}_l = \bigcup_{b \in l} \texttt{B_Ambiguous}_b$$

$$\texttt{L_Promotable}_l = \texttt{L_Explicit}_l - \texttt{L_Ambiguous}_l$$

$$\texttt{L_Lift} = \begin{cases} \texttt{L_Promotable}_l & \text{if l is an outermost loop} \\ \\ \texttt{L_Promotable}_l - \texttt{L_Promotable}_{surrounding-loop(l)} & otherwise \end{cases}$$

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、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 3.1 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 Infomation						
	B1	B2	B3	B4	B5	B6
B_Explicit	p,q	a	b	a	c	
B_Ambiguous		b	c			

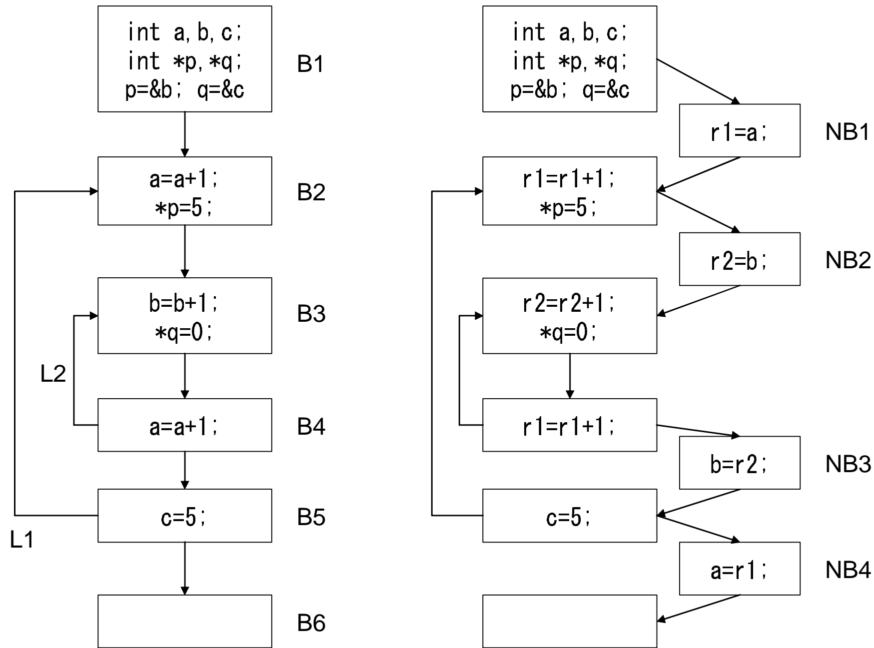

図 3.1: レジスタプロモーションの例

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_Explcit` 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 Infomation					
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4章　実装の準備、設定

4.1　COINSについて

4.1.1　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標として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る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-Project]。

4.1.2　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**(immediate 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。これらの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4.1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 represantation, LIR) と呼ぶ。

COINSのソースはすべてJava言語で書かれている。

4.2　グローバル変数のレジスタプロモーション

4.2.1　目標

COINSにはメモリにしまわれている値をレジスタに促進する最適化を行う部分が既に組み込まれている。しかし、そこで対象としている値はローカル変数のみであり、あいまいな参照などの解析が必要ない範囲で適用されている。具体的には、おもにアドレスを取られていないローカル変数を対象としている。（詳しくはCOINSの仕様書のバックエンド部 [森公一郎] を参照）本論文では、入力言語としてC言語を対照とし、特にC言語のグローバル変数についてのレジスタープロモーションを目標とする。なお、グローバル変数は大域変数とも呼ばれ、プログラムの中のもっとも外側のブロックで宣言された変数であり、すべての手続き（関数）においてで有効ある。つまりプログラムのどこからでも参照・更新することができる。これに対して、宣言されたブロック内だけで有効な変数をローカル変数、または局所変数という。

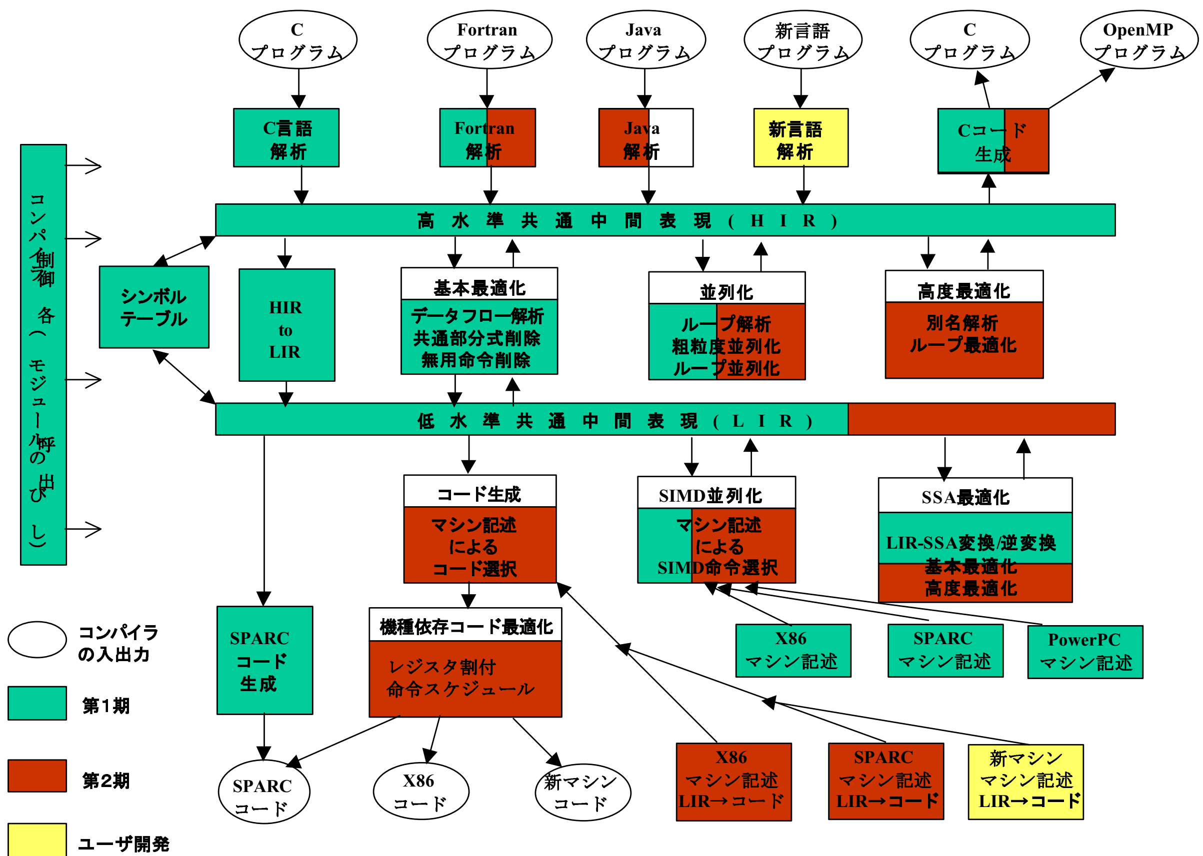

図 4.1: 並列化コンパイラ向け共通インフラストラクチャ(COINS) 概念図

4.2.2　レジスタプロモーションの精度

以下、グローバル変数のみを対象として話を進める。COINS のバックエンドでは、手続きごとに最適化などの処理が行われるため、現在処理中の手続き以外の部分で設定された変数同士の関係などに関する情報は得ることができない。よってある手続きに関してレジスタプロモーションを行うとき、その手続きの外部の情報は一切ないことを前提とする。レジスタプロモーションは L_Ambiguous$_l$ の取り方によって、実装の難易度と実行時の精度が異なる。ループ l 内でのあいまいな参照をすべて正確に発見するのは、ループしだいではかなり困難な作業である。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、ポインター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 内で使われるタグ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。(2) に関して、具体例を用いて説明する。下のようなコードを考えてみる。

```
int *p;
int a;
p = &a;
f(){
  int b = 1;
  a = 0;
  for(int i = 1; i <= 10; i++){
     a = a + i;
     *p = b + 1;
  }
}
```

for 文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 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、for 文の中で*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 にしまうとすると、for 文は次のように書き換えられる。

```
r = a;
for(int i = 1; i <= 10; i++){
  r = r + i;
  *p = b + 1;
}
a = r;
```

*p = b + 1 はポインタ p の指す場所つまりタグ a のメモリ位置に b + 1 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 for 文を出て、レジスタからメモリ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 = b + 1 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。これより、ループ l 内にポインタ間接参照がある場合、それが l の中のタグを参照している危険性があるのでループ l 内で使われるすべてのタグをあいまいな参照を受けるものとして扱う。

よって、本論文においては、ループ l において、call 文または間接的な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 $\texttt{L_Ambiguous}_l$ 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。

第 5 章　設計と実装

5.1　はじめに

グローバル変数のレジスタープロモーションは、COINS にその機能を追加する形で実装する。COINS のバックエンドに最適化器の役割を担う class ファイルを集めたフォルダ opt があり、ここに新しくレジスタプロモーションを行う class または java ファイル Regpromote.java と RegPromote.class を追加する。

5.2　COINS のバックエンドについて

COINS のバックエンドでは、中間コードの構成要素に関して、構成要素それぞれに対応するクラスが用意されており、そのクラスのインスタンスとして表現される。例えば中間コード内の手続きは、これに対応する Function クラスのインスタンスとして扱われる。

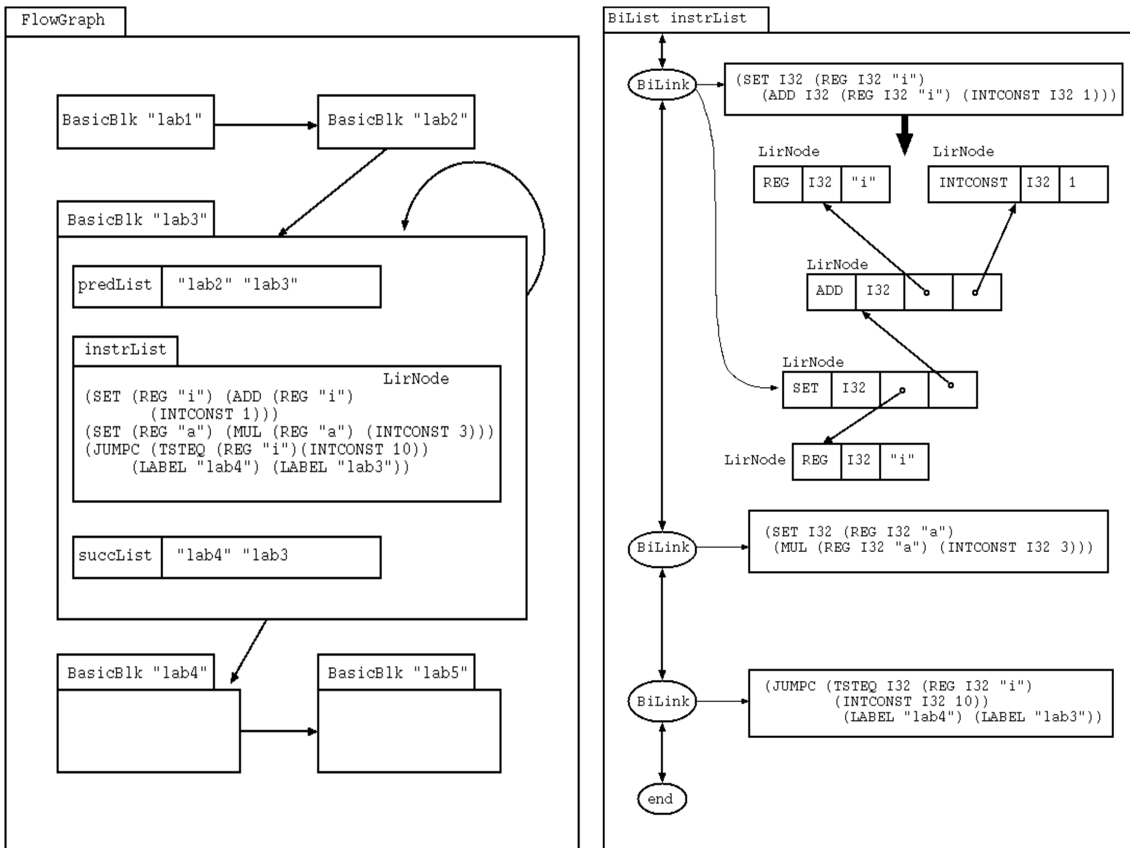

図 5.1: LIR での制御フローグラフと基本ブロックの例

LIR において手続き内部の制御フローグラフを表す FlowGraph クラスの内部の様子を図 5.1 に示

した。左側が FlowGraph クラスの内部を示している。FlowGraph クラスのフィールドには、この制御フローグラフに属する基本ブロックを表す BasicBlk クラスが保存されている。BasicBlk クラス内には、そのブロックで実行される命令を保存する instrList というリスト、このブロックの先行ブロックを保存する predList、後続ブロックを保存する succList などがある。

右側は基本ブロック内の命令のリスト instrList の内部を示している。instrList の要素一つ一つが、命令文となっている。例えば図の例で言えば、一番目の要素は

```
(SET I32 (REG I32 "i") (ADD I32 (REG I32 "i") (INTCONST I32 1)))
```

という命令である。これは変数 i の値がしまってあるレジスタの中身（つまり i の値）に 1 を足しこむというものである。(REG I32 "i") は変数 i の値をしまったレジスタの中身（つまり i の値）、(INTCONST I32 1) は定数 1 を表す。ADD は二つの引数を足した結果を表す。SET は第一引数に、第二引数の値を代入するという命令である。見て判るように、一つの命令文は木構造になっている。先ほどの命令の木構造は、図の右側の中ほどに示されている。木のノード一つ一つは、命令文を構成する要素を表す LirNode というクラスのインスタンスである。

また、図では示されていないが、(STATIC I32 "a")、(FRAME I32 "b") はそれぞれ、グローバル変数 a のメモリアドレス、ローカル変数 b のメモリアドレスを表す LirNode である。このアドレスにしまわれている値を取り出すには、メモリから値を取り出す命令 MEM を使って (MEM I32 (STATIC I32 "a")) のようにする。さて、グローバル変数のアドレス、ローカル変数のアドレス、レジスタ、それぞれの LirNode には symbol(シンボル) と呼ばれる、Symbol クラスのインスタンスが割り当てられていて、それぞれのフィールドにしまわれている。簡単に言ってしまえば、symbol はタグを表すものである。例えば先ほど出てきた命令には (REG I32 "i") という LirNode が二つある。この二つは LirNode クラスの異なるインスタンスであるが、同じレジスタの値を表す。そのため、どちらの (REG I32 "i") の symbol も、同一の Symbol クラスのインスタンスになっている。同一の変数の値がしまわれているレジスタには同一の Symbol のインスタンスを対応させるというわけである。(STATIC I32 "a")、(FRAME I32 "b") といった LirNode に関しても同様で、(STATIC I32 "a") が命令で複数回使われていても、同じグローバル変数 a のアドレスを表すなら、どの (STATIC I32 "a") の symbol も、Symbol クラスの同じインスタンスである。

後にも述べるが、本研究の実装では $\texttt{L_Ambiguous}_l$ や $\texttt{L_Explicit}_l$ などには、Symbol クラスのインスタンスが保存される。

5.3 全体構造

Regpromote.java 内のクラスは以下の 2 つを用意した。同時に、フィールドとメソッドの概要も示す。レジスタプロモーションを行うクラス RegPromote は以下の要素で構成される。

public class RegPromote	
フィールド	
BiList LoopList	手続き内のループを表す RPloop のインスタンスをしまうリスト
メソッド	
public void doIt	レジスタプロモーションを実行するメソッド
public BiList compMember	ループの member のリストを完成させるメソッド

ループを表すクラス RPloop は以下の要素で構成される。

public class RPloop	
フィールド	
public BasicBlk head	ループのヘッダとなるブロック
public BiList member	ループの要素ブロックのリスト
public BiList srndLoop	このループを含むループのリスト
public BiList exitList	ループからの脱出先のブロックのリスト
public int nestLevel	このループのレベル
public Function f	このループが存在する手続き（関数）
public BiList L_Explicit	このループの L_Explicit
public BiList L_Ambiguous	このループの L_Ambiguous
public BiList L_Promotable	このループの L_Promotable
public BiList L_Lift	このループの L_Lift
private BiList DSlist	実装の便宜上使うリスト
private boolean isAmb	このループが促進の対象になるかどうか
public RPloop	コンストラクタ
メソッド	
public void addExit	exitList を完成させるメソッド
public void getGV	L_Explicit と L_Ambiguous より L_Promotable を計算するメソッド
public void searchGV	命令をたどり、L_Explicit と L_Ambiguous を完成させるメソッド
public void getLIFT	L_Lift を完成させるメソッド
public void insertNewInst	このループの入り口、出口に新しいブロックを挿入しメモリ、レジスタ間のロードやストアの命令を挿入するメソッド
public LirNode getToRegInst	レジスタとメモリ間の命令を生成して返すメソッド
public void PreCTR	ループ内のメモリ参照をレジスタ参照に置き換えるメソッド
public void changeToReg	ループ内のメモリ参照をレジスタ参照に置き換えるメソッド

5.4 各段階の説明

レジスタプロモーションは、クラス RegPromote の doIt メソッドで実行される。このメソッドの中でクラス RPloop などのメソッドを使いつつ処理を進めていくことになる。doIt メソッドの内部を順を追って説明する。

5.4.1 RPloop のインスタンスの生成

まずはじめに手続き内の自然ループを見つける。COINS にはループ解析を行うクラス LoopAnalysis が用意されているのでこれを利用する。LoopAnalysis は、ヘッダが複数あるようなループ（つまり自然ループでないもの）も見つけるので、対象とするものをあらかじめヘッダが一つのものに絞っておく。自然ループを見つけたら、そのループについて、RPloop のインスタンスを作り、LoopList に追加する。次に LoopList のすべてのループについて、ループの要素ブロック (member)、このループを含むループ (srndLoop)、ループからの脱出先のブロック (exitList) の情報を追加し、レジスタプロモーションの準備を整える。

5.4.2 促進可能なタグの計算

ループの準備ができたら、各ループの getGV メソッドを実行する。第４章においてアルゴリズムの説明ではこのメソッドは各ブロックについて $\texttt{B_Explicit}_b$、$\texttt{B_Ambiguous}_b$ を求めると書いたが、実装ではこれを行わず、ループ内の命令を直接たどることで $\texttt{L_Explicit}_l$ と $\texttt{L_Ambiguous}_l$ を計算し、その結果より $\texttt{L_Promotable}_l$ を求める。ループの要素ブロックの命令をたどるところまでをこのメソッドで担い、$\texttt{L_Explicit}_l$、$\texttt{L_Ambiguous}_l$ を実際に求める仕事は searchGV メソッドに丸投げされる。searchGV メソッドは引数の設定によって、

（1）命令の中に `call`(手続き呼び出し) やメモリへの間接参照がないかどうかの調査

（2）ループ内のタグの $\texttt{L_Explicit}_l$ または $\texttt{L_Ambiguous}_l$ への追加

という２つの機能を使い分ける。メソッド内の構造は下に示したようになっており、`instsrch` が true であれば (1) の機能を、false であれば (2) の機能を使う。

```
public void searchGV(LirNode node, boolean allamb, boolean instsrch){
  if(instsrch){
      関数呼び出しかメモリへの間接参照があればこのループの isAmb を true にする。
  }

  if(node.opCode == Op.MEM && !instsrch){
    LirNode mem = node.kid(0);
      if(mem.opCode == Op.STATIC){ // グローバル変数への直接参照
        if(!allamb){
            (引数 allamb には必ず isAmb が入るような仕様になっている。)
            allamb が false であればこのループのすべてのグローバル変数
            のタグが L_Explicit_l に加えられる。
        }else{
            allamb が true なので、このループのすべてのグローバル変数
            のタグが L_Ambiguous_l に加えられる。
        }
      }
   }

   再帰で命令をたどる。
   int n = node.nKids();
   for(int i=0; i<n; i++){
     LirNode x = node.kid(i);
     searchGV(x, allamb, instsrch);
   }
}
```

まず、`instsrch` を true にしてループ内の全命令をたどらせ、`call`(手続き呼び出し) やメモリへの間接参照を探す。その結果に応じて `isAmb` を書き換える。

次に `instsrch` を false にしてループ内の全命令をたどらせ、タグの $\texttt{L_Explicit}_l$ または $\texttt{L_Ambiguous}_l$ への追加を行うという流れになる。

isAmb が false つまりループ l 内に手続き呼出しまたは間接参照があれば、このループのタグは、すべてあいまいな参照を受けると判断されるため促進されない。よって $\texttt{L_Explicit}_l$ を求めても無駄になってしまう。同様に isAmb が true であれば、このループのすべてのタグは促進されるので $\texttt{L_Ambiguous}_l$ を求めても無駄になってしまう。よって isAmb が true であれば $\texttt{L_Explicit}_l$ の追加のみ、false であれば $\texttt{L_Ambiguous}_l$ の追加のみを行うように実装した。

なお、searchGV メソッドにおいて命令文は再帰でたどっていく。

LIR における間接参照の判定についても述べておかなくてはならない。前提として、スカラー値、ポインター、配列のみを対象とする (それで十分なはずである)。LIR では、メモリにしまわれている値を参照する時 MEM という命令を使う。例えば、グローバル変数 a の値は

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

という命令で参照できる。I32 というのは参照する値の型、(STATIC "a") はグローバル変数 a のアドレスを示すものでありローカル変数なら (FRAME "a") のようになるが、ここではあまり重要ではない。LIR での間接参照をポインタを例にとると、ポインタ p の参照先にしまわれている*p の値は、

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

のように参照される。また*(p+8) のようなものは

```
(MEM I32 (ADD((MEM I32 (STATIC "p"))(INTCONST I32 8))))
```

のように参照される。配列についても述べなくてはならない。例えば配列 a[10] があったとき、a[0] から a[9] に連続したメモリ領域が割り当てられ、さらにその先頭 a[0] を指すポインター a のメモリ領域が割り当てられる。よって、例えば１要素が４ byte とすると、a[1] の値は、

```
(MEM I32 (ADD((MEM I32 (STATIC "a"))(INTCONST I32 4))))
```

のように参照される (１つ上の例と同じ型になる)。ここで、(INTCONST I32 4) の部分の値を自由に変えると、配列に割り当てられた領域を飛び出し、グローバル変数にアクセスする可能性も出てくるが、割り当てられた領域内でのみ動くことは前提としてよい。よって、上のように MEM の子ノードとして ADD が来ていれば、それは配列の要素の値への参照であると判断できる。さらに、配列の要素の参照は間接参照として扱う必要はない。

以上のことから、命令をたどっていって、MEM を見つけたとき、その引数が MEM であった場合これは間接参照であると判断する。

このあと getGV メソッドでは $\texttt{L_Promotable}_l$ を求める。これは単に $\texttt{L_Explicit}_l$ の要素で $\texttt{L_Ambiguous}_l$ に含まれていないものを $\texttt{L_Promotable}_l$ に追加していくだけである。各ループに getGV メソッドを適用した後で、各ループにおいて getLIFT メソッドにより $\texttt{L_Lift}_l$ を求める。getLIFT の内部も単純で、$\texttt{L_Promotable}_l$ の要素のうち、ループ l のサラウンディングループの $\texttt{L_Promotable}_l$ に含まれていないものを $\texttt{L_Lift}_l$ に追加していく。

5.4.3 命令の書き換えと追加

各ループの $\texttt{L_Lift}_l$ が求まれば、後はその情報に基づいて命令を書き換えればよい。

insertNewInst メソッドはループの入り口にメモリからレジスタへのロードを、出口にレジスタからメモリへのストアを挿入する。まずループのヘッダの先行ブロックを一つ作り、それまでヘッダにつながっていた有向辺のうちループの外から来たものに関して行き先を新しい先行ブロックに変更する。この先行ブロックで $\texttt{L_Lift}_l$ に含まれる symbol のアドレスにしまわれている値をレジスタにロードする。例えばタグ a の symbol が $\texttt{L_Lift}_l$ に含まれていた場合、

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

という命令をこの新しいブロックに挿入する。(REG I32 "a") はタグ a の値をロードするレジスタである。この命令は getToRegInst メソッドで作られる。$\texttt{L_Lift}_l$ の要素すべてに関してロードの命令を作り、新しいブロックに挿入する。

getToRegInst メソッドは、メモリからレジスタへのロード、レジスタからメモリへのストアの命令を生成して返すメソッドである。引数によってどちらの命令を作るか指定できるようになっている。

ループの出口に関しても insertNewInst メソッドは同じような作業を行う。exitList の要素であるブロックそれぞれについて、先行ブロックを一つずつ作り、exitList の要素の先行ブロックのうち、ループの要素であるものからの有向辺の行き先を新しく作った先行ブロックに変更する。新しく作ったブロックには、上記と同じように getToRegInst メソッドを使ってレジスタからメモリへのストア命令を生成し挿入する。

新しいブロックの挿入に関しては、少し説明をしておく。また図 5.2 に様子を示しておいた。COINS バックエンドの FlowGraph クラスには、「指定したブロック B の直前に新しいブロックを挿入する」というメソッドが用意されている。このメソッドは、先行ブロックを持たず、後続ブロックがブロック B だけであるような新しいブロックを作る。まず、このメソッドを使って図の左のフローグラフのように header と 2 つの exit の直前にブロックを挿入する。そして、header については header の先行ブロックのうちループ外にあるものからの有向辺を、新しいブロックに向かわせる。exit については exit の先行ブロックのうちループの要素であるものからの有向辺を新しいブロックに向かわせる。これで図の右のような挿入後のフローグラフが完成する。

preCTR はループ内のメモリ参照をレジスタへの参照に置き換えるメソッドだが、置き換えの仕事は changeToReg メソッドに丸投げされている。changeToReg メソッドでは、$\texttt{L_Lift}_l$ の情報をもとに該当するメモリ参照の命令 MEM 命令をレジスタ参照 REG 命令に置き換える。

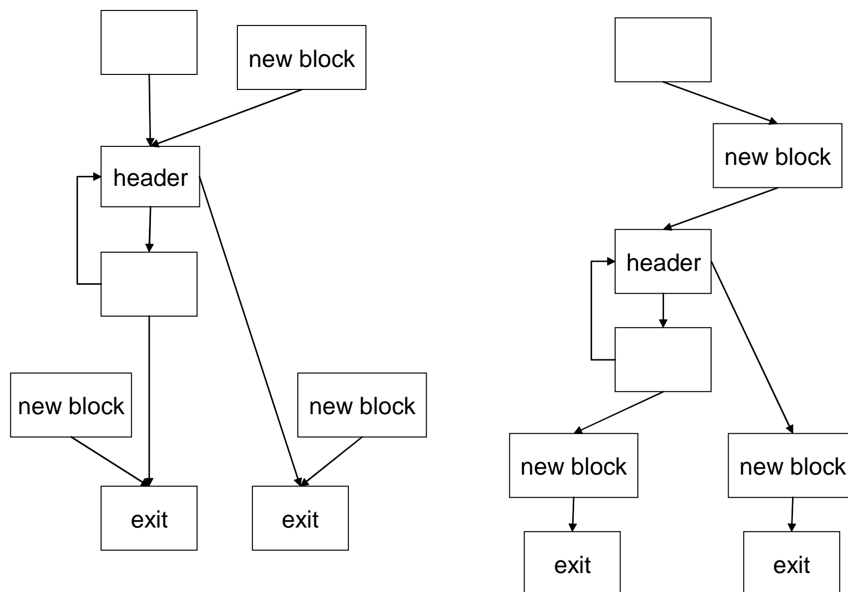

図 5.2: 新しいブロックの挿入

第6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使ってレジスタプロモーションを行い、レジスタプロモーションしなかった場合と比較する。

6.1　実行環境とテストプログラム

実験は、Sun Microsystems の、Sun Blade 1000 で行った。この主な仕様は、表 6.1 の通り。

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 6.1: 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。
COINS の開発、テストに使われているプログラムより以下の６つ。

> ヒープソート (heap.c)、１４女王問題 (queen.c)、シェルソート (shell.c)、挿入ソート (InsertionSort.c) 選択ソートプログラム (SelectionSort.c)、素数算出プログラム (tpprime.c)

SPEC CPU2000 のベンチマークプログラムより以下の９つ。

> 164.gzip, 197.parser, 254.gap, 256.bzip2, 300.twolf, 177.mesa, 179.art, 183.equake, 188.ammp

COINS のプログラムについては実験の各項目においてそれぞれ５回ずつ、SPEC のプログラムについてはそれぞれ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。

6.2　実行結果と考察

6.2.1　COINS の開発プログラムの結果

レジスタプロモーションをした場合、レジスタプロモーションをしていない場合の実行時間は次のようになった。数値の単位は秒である。

実行結果		
	レジスタプロモーションあり	レジスタプロモーションなし
ヒープソート	83.34	83.19
１４女王問題	86.99	87.03
シェルソート	90.43	90.49
挿入ソート	224.94	224.82
選択ソート	323.28	321.73
素数算出	227.51	355.20

結果として、素数算出の実行時間は 30%以上短縮されたが、それ以外については目立った差はなかった。

COINS の gcc によってはきだされた SPARC のコードを調べると、本研究で実装したレジスタプロモーションが行われていたのは、ヒープソート、１４女王問題、素数算出の３つであった。この３つについて、促進されたタグの個数、SPARC のコードでメモリ参照からレジスタ参照に置き換えられた箇所の個数を調べると下の表のようになった。

結果の比較		
	促進されたタグ数	置き換わり箇所数
ヒープソート	3	4
１４女王問題	3	12
素数算出	8	35

ヒープソートと１４女王問題については、レジスタに置き換えられた箇所はすべて小さいループであり、反復回数も１０から２０回程度のものであった。よって、この２つのプログラムで実行時間が短縮されなかったのはほぼ当然の結果といえる。素数算出においては２重ループが２つ、重なっていないループが１つあったが、２重ループ２つのうち１つと、重なっていないループについてレジスタプロモーションが適用されていた。２重ループのほうでは、ループ内でのグローバル変数の使用頻度が高く、促進されているグローバル変数は７つと多かった。この時点で、レジスタプロモーションが適用された他のプログラムよりも、適用の度合いにかなり差がある。さらに外側のループに関しては反復回数が１００万回を超えるものであったため、これらが素数算出の実行時間の改善の要因であろう。

次に、これらのプログラムを C 言語コンパイルの gcc でオプション-O2 をつけてコンパイルして実行してみた。このオプションは gcc であらかじめ用意されているさまざまな最適化をかけるためのもので、この最適化はかなり強力であるがレジスタプロモーションは行われない。レジスタプロモーションのみをかけた先ほどのデータと比較すると次のようになる。数値の単位は秒である。

実行結果２		
	レジスタプロモーションのみ	gcc -O2 のみ
ヒープソート	83.34	73.41
１４女王問題	86.99	74.45
シェルソート	90.43	74.59
挿入ソート	224.94	125.39
選択ソート	323.28	184.77
素数算出	227.51	256.63

上５つの実行時間は gcc -O2 の方が、実行時間が大きく短縮される。しかし素数算出については、レジスタプロモーションのみの方が 10%ほど早くなった。これはレジスタプロモーションが、十分に適用されるようなプログラムではいかに強力な最適化であるかということを示しているといえる。

通常、最適化は複数のものを組み合わせて使うので、複数の最適化を行ったものについても調べてみた。

SSA 変換 → 定数伝播 → 共通部分式除去 → SSA 逆変換

という順序の SSA 最適化にさらにレジスタプロモーションをかける場合とかけない場合で実行時間を計測した。

実行結果３		
	SSA 最適化 + レジスタプロモーション	SSA 最適化のみ
ヒープソート	82.29	84.30
１４女王問題	81.22	81.47
シェルソート	76.74	76.41
挿入ソート	225.49	224.63
選択ソート	359.35	360.16
素数算出	227.20	349.85

依然として、素数算出については、レジスタプロモーションを行う方が早い。しかし、他のプログラムでは SSA 最適化をかけても、レジスタプロモーションをする場合としない場合の関係はほぼ保たれたといえる。

6.2.2　SPEC のベンチマークプログラムの結果

SPEC のベンチマークプログラムでレジスタプロモーションを行ったものと行わなかったものの結果は次のようになった。

実行結果		
	レジスタプロモーションあり	レジスタプロモーションなし
164.gzip	870	873
197.parser	874	922
254.gap	759	733
256.bzip2	1052	1052
300.twolf	1339	1218
177.mesa	1122	1120
179.art	733	734
183.equake	1115	1121
188.ammp	2113	2114

いくつかのプログラムで 1~5%程度の実行時間の改善が見られた。ログを調べたところ、改善のほとんどゼロに等しいものに関しては、促進された値の個数も著しく少なかった。300.twolf においては 10%程度実行時間が長くなった。このプログラムにおいては大変多くの値が促進されているのだが、吐き出されたコードを調べたところ、１つのループに対して最大で２０個以上もの値を促進していた。このためレジスタ圧力が上がり過ぎ、スピルが起こったためにこのように実行時間が遅くなってしまったと考えられる。

6.2.3　まとめ

総じて見ると、レジスタプロモーションは、実行頻度の高い部分に適用された場合には絶大な効果があることがわかる。その反面、一度に促進する値の個数が多すぎるとレジスタ圧力が上がりすぎ、実行時間を長くしてしまうという一面もあるようだ。本研究では、対象をグローバル変数のみに絞るなどいくつか制限を加えたため、レジスタプロモーションの適用範囲が狭まり、改善の見られないケースが多かった。次節で述べるような拡張を行えば、飛躍的に効果が上がると思われる。

6.3　今後の課題

今回の研究では、レジスタプロモーションの適用範囲をかなり絞った実装を行ったが、これでは特定の条件を満たした場合にしか役に立たない。レジスタプロモーション自体は有効な最適化であるから、今後次に述べるような改善を加えたい。

6.3.1　手続き間への拡張

本研究では手続き呼び出しがあるループに関しては、レジスタプロモーションを適用しなかった。呼び出し先の手続き内を解析することで、手続き間まで拡張したレジスタプロモーションが行えそうである。手続き呼び出しを含むループは多いはずであるから、これは今後是非実現すべき項目である。しかし、手続きごとに最適化を行う COINS では困難といえる。

6.3.2　ポインタ解析を行う促進

ポインタ等の参照関係の解析をポインタ解析というが、ポインタ解析を行いその結果を使えば、ループ内に間接参照があってもレジスタプロモーションを適用できる場合がある。以下の例では、ポインタ解析を行えば、ポインタは for ループ内で常に a のアドレスを指すことをつきとめるので、こ

のループで a にレジスタプロモーションが適用できる。

```
int *p;
f(){
   p = &b;
  int b = 1;
  int a = 0;
  for(int i = 1; i <= 10; i++){
     a = a + i;
     *p = b + 1;
  }
}
```

これにより、ローカル変数にもレジスタプロモーションが適用できるようになり、今までの素朴なレジスタ割り当てのみの場合より幅が広がる。もちろんグローバル変数についても、本研究で実装したものよりさらに多くの値の促進が期待できる。

6.3.3 促進個数の制限

一度に促進する値の数が多すぎると、実験でも起こったようにレジスタ圧力が上がりすぎ、実行時間を逆に長くしてしまう。これを回避するために適度に促進個数を制限する必要がある。

第 7 章　関連研究

本章ではレジスタプロモーションに関して行われている研究について述べる。

7.1　手続き間への拡張

６章の「今後の課題」でも述べた手続き間に拡張されたレジスタプロモーションについて服部らの論文 [服部直哉他 2000] で述べられている。この論文で扱われているアルゴリズムは、手続き間のポインタ解析をすることによる促進を行うものである。

7.2　部分冗長除去の利用

Lo、H. らの論文 [Lo et al. 1998] では静的単一代入 (SSA) 形式での部分冗長性除去 (PRE) を用いたレジスタプロモーションについて述べられている。レジスタへのロードとレジスタへのストアに着目した冗長性除去を行い、値更新をレジスタ上で行うようにするというものである。

7.3　SSA 形式での促進

A.V.S Sastry らの論文 [Sastry and Ju 1998] では、SSA 形式において、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レジスタプロモーションが紹介されている。

第8章　おわりに

本研究ではグローバル変数のレジスタプロモーションをコンパイラ・インフラストラクチャ上で実装しさらに評価を行った。一部のプログラムにおいては実行時間の改善に成功した。レジスタアクセスをメモリアクセスに置き換えるという最適化はやはり効果が大きく、レジスタプロモーションは実行頻度の高い部分で適用できれば大きな実行時間の改善が期待できるようである。

しかし、本研究では対象をグローバル変数とするなどの制限を加えたため、一部のプログラムにおいてはレジスタプロモーションがほぼ適用されず、実行時間の改善が実質上ないものも多かった。また、レジスタプロモーションを一度に適用しすぎたために、レジスタ圧力が上がり、実行時間が遅くなったものもあった。これらについては6章でも述べたように適用範囲を広げていきたい。

謝辞

本研究に進めるにあたり多大なる御指導御鞭撻を頂いた、東京工業大学 数理・計算科学専攻教授の佐々政孝先生に心から御礼申し上げます。

　また本研究を進めるにあたって、お忙しい中多くのご指導、助言をしてくださった溝渕裕司先輩、須藤大二郎先輩、伊藤陽先輩、同級生の佐原聡一郎君、そして研究に疲れたときよく話し相手になって下さった方玲さんに心から御礼申し上げます。

　実装にあたって、細かい質問にも丁寧に答えてくださった LSI Japan の森公一郎さんに心から御礼申し上げます。

参考文献

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman 1985. *Compilers : Principles, Techniques, and Tools.* Addison Wesley.

COINS-Project "COINS-Project homepage.". http://www.coins-project.org/.

伊藤陽 2004. 「コンパイラ · インフラストラクチャ上の部分冗長除去の実装と評価」東京工業大学 情報科学科 卒業論文.

Lo, Raymond, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu 1998. "Register promotion by sparse partial redundancy elimination of loads and stores." *SIGPLAN Not.* Vol. 33. No. 5. pp. 26–37.

Lu, John and Keith D. Cooper 1997. "Register promotion in C programs." In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation.* ACM Press.

中田育男 1999. 『コンパイラの構成と最適化』朝倉書店.

佐々政孝 1989. 『プログラミング言語処理系』岩波書店.

Sastry, A. V. S. and Roy D. C. Ju 1998. "A new algorithm for scalar register promotion based on SSA form." In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation.* ACM Press.

立川英 2002. 「循環リモート属性文法による部分冗長性除去の記述」東京工業大学 情報科学科卒業論文.

森公一郎"COINS-Project Backend Part.". http://soukou.cs.uec.ac.jp/ kmori/.

服部直哉・飯塚大介・坂井修一・田中英彦 2000. 「コンパイラによるロード・ストア負荷の軽減」『ハイパフォーマンスコンピューティング　８２－１９』.

立川英 2004. 「静的単一代入形式上の部分冗長性除去」東京工業大学 情報理工学研究所数理・計算科学専攻　修士論文.