

レジスタプロモーションによるコード最適化の実装と評価

及川 宗明 佐々 政孝

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乗せる。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、どうしてもメモリアccessを必要とする部分が存在する。このような変数を、アクセスされるメモリ上の位置が明確な範囲でレジスタに格上げする最適化をレジスタプロモーション [2] と呼び、これを COINS [1] 上に実装した。

1 はじめに

無限個のレジスタを仮定する仮想レジスタを使ったとしてもレジスタに格上げすることができない変数として

1. メモリアドレスを使用される変数
2. グローバル変数

が挙げられる。このような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をレジスタプロモーションという。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。以前、COINS [1] 上でのレジスタプロモーションの研究が行われている [4]。しかし、グローバル変数を対象としたレジスタプロモーションしか適用できていない。本研究ではメモリアドレスを使用される変数のレジスタプロモーションを COINS 上に実装することで実

行時間の改善に成功した。一方、レジスタの個数には限りがあるため、素朴な方法では大規模なプログラムでスピルアウトが頻繁に起こってしまう。スピルアウトを防ぐために別名解析 [5] の情報を用いてレジスタの節約を行った。

2 準備

2.1 COINS

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開発された、並列化コンパイラ向け共通インフラストラクチャである。COINS のバックエンドはフロントエンドが出力した低水準中間コードファイル (LIR) を読み込んで機械語を生成する。本研究のレジスタプロモーションは LIR 上に実装する。

COINS の LIR では、

1. メモリアドレスを取られている変数
2. グローバル変数

以外の変数を仮想レジスタに割り付ける。その後、最適化など種々の処理を行ったあと、レジスタ割り当てにより、仮想レジスタを実レジスタに割り当てる。

2.2 別名解析

本研究では、以前吉羽さんが COINS 上に実装した別名解析 [5] [6] を使用する。そこで、吉羽さんが実

Implementation and Evaluation of a Code Optimization by Register Promotion.

Muneaki Oikawa, 東京工業大学情報理工学研究所, Dept. of Mathematical and Computing Sciences, Tokyo Institute of the Technology,

Masataka Sassa, 東京工業大学情報理工学研究所, Dept. of Mathematical and Computing Sciences, Tokyo Institute of the Technology,

装した別名解析について説明する。

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

1. ポインタ型
2. 関数呼び出し

などがあげられる。

まず、グローバル変数の関数呼び出しについて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() {
    int i;
    for(i = 0; i < 10; i++) {
        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る。

次に、ポインタについて述べる。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。

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

main() {
    p = &a;
    p *= 1;
}
```

別名解析とは、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。また、別名解析には flow-sensitive なものと flow-insensitive なものが

存在する。

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、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、精度は低い

本研究では flow-insensitive な別名解析 [6] [5] を採用した。

2.3 レジスタプロモーションのアルゴリズム

レジスタプロモーション [3] [4] [6] は以下のように進めていく。ここで出てくる「タグ」とは命令に使われるメモリ位置の字面上の識別名のことである。

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。あいまいな参照とは間接的な参照のことである。先ほどの関数呼び出しに対する別名解析の例では globalVariable があいまいな参照を受ける。また、ポインタ型に対する別名解析の例では a があいまいな参照を受ける。

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、sur-loop(l) とはループ l のサラウンディングループ (取り囲むループ) のことである。

$$L_Explicit_l = \bigcup_{b \in l} B_Explicit_b$$

$$L_Ambiguous_l = \bigcup_{b \in l} B_Ambiguous_b$$

$$L_Promotable_l = L_Explicit_l - L_Ambiguous_l$$

$$L_Lift_l =$$

$$\left\{ \begin{array}{l} L_Promotable_l \\ \text{(if } l \text{ is an outermost loop)} \\ \\ L_Promotable_l - L_Promotable_{sur-loop(l)} \\ \text{(otherwise)} \end{array} \right.$$

4. コードを書き直す 各々のループについて

$L_Promotable_l$ 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、そこで仮想レジスタを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は、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 $L_Promotable_{sur-loop(l)}$ を除いた集合になる。

3 本研究における設計と実装

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 [3] [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。

3.1 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 [3] [4] [2]。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)、(2) の状況が生じた場合、仮にレジスタに格上げされた変数があったとすると、その値が壊されてしまう可能性がある。そこでループ内に出てくる (1) と (2) のグローバル変数のタグはすべて $B_Ambiguous_l$ の要素に加える。

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;
    for(i = 0; i < 10; i++) {
        a = a + i;
    }
}
```

```

    square();
}
}

```

for ループの部分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 にしてしまうとすると、ループは次のように書き換えられる。

```

r = a;
for(i = 0; 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。

ここでは (2) の説明は省略する。

3.2 ポインタがある場合のレジスタプロモーション

本節では、ポインタがある場合の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がある場合の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、そのループ内のグローバル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;
int b = 0;

f() {
    p = &a;
    int i;
    for(i = 0; 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 LIR 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 あるいは b 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 の指す内容は変数 b の別名

に書き換えられたにも関わらず、ポインタ p の指す内容は変数 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。

3.3 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] [5] 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くなってしまうことをいう。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;
int *q;
f() {
  p = &a;
  q = &a;
  int i;
  for(i = 0; 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;
int *q;
int *r;

f() {
  p = &a;
  q = &a;
  r = &a;
  int i;
  for(i = 0; 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ため、同じレジスタに格上げすることができる。

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。

4 実験結果と考察

本節では実装したレジスタプロモーションの実験結果とそれに対する考察を行う。

実験環境は

テストプログラムには、SPEC CPU2000 ベンチマークを用いた。

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、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のそれぞれを比較した結果は、図 1 のようになった。

図 1 を見てわかるように全体的に実行時間が改善

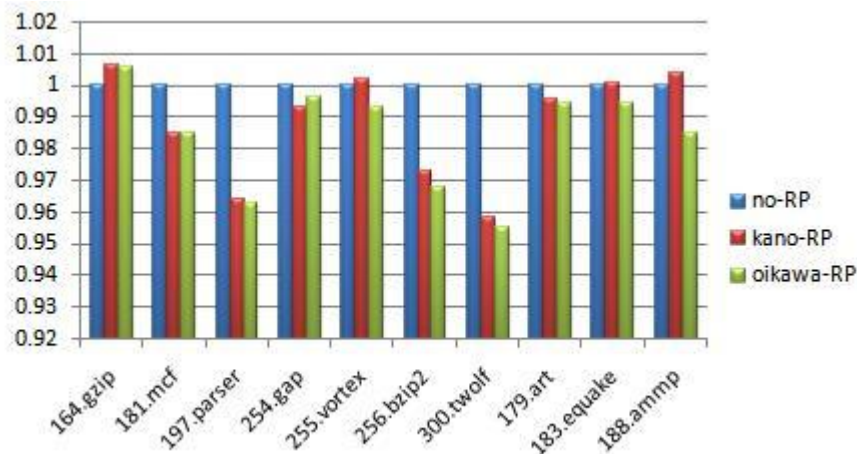


図 1 実行時間の比較

アーキテクチャ	Superscalar SPARC Version 9
プロセッサ種別	750MHz UltraSPARC
一次キャッシュ	64KB Data 32KB Instruction
二次キャッシュ	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

されていることがわかる。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、今回のポインタに対す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と思われる。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いうこともあるからである。

本研究のレジスタプロモーションでは別名解析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ことにより、今までレジスタプロモーションを適用できなかったグローバル変数、ポインタ変数をレジスタに格上げすることが可能になった。よって、本研究でポインタに対するレジスタプロモーションによる効果が示された。

5 まとめと今後の課題

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見られた。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ふ。よって、実行される回数が少ない変数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 本論文の初期の版について議論していただいた研究室の皆様へ感謝する。本研究の一部は科学研究

費補助金の援助を受けた。

参考文献

- [1] COINS Project:Coins homepage. <http://www.coins-project.org/>.
- [2] Cooper,K.D. and Lu,J.:Register Promotion in C Programs. PLDI, pp. 308-319, 1997.
- [3] 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論文, 2005. <http://www.is.titech.ac.jp/~sassa/lab/papers-written/kano-thesis.pdf>.
- [4] 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).
- [5] Steensgaard,B.: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.
- [6] 吉羽和之:型推論に基づく手続き間ポイント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論文, 2006. <http://www.is.titech.ac.jp/~sassa/lab/papers-written/9927000.pdf>.