Flow-insensitive な別名情報と制御フロー構造を用いた最適化対象の拡大

狩野　祐介 †　　佐々　政孝 †
Yuusuke KANOU　　Masataka SASSA
† 東京工業大学 大学院情報理工学研究科 数理・計算科学専攻
Dept, of Mathematical and Computing Sciences, Tokyo Institute of Technology
{ kano2, sassa }@is.titech.ac.jp

メモリ参照をレジスタ参照に置き換えることは、実行時間の改善という点において有用な最適化であるが、別名の存在がその妨げとなり、置き換えを保守的なものにとどめてしまうことがある。別名解析を行うことでこの問題を解決できるが、別名解析には大きく分けて flow-sensitive なものと、flow-insensitive なものの 2 種類がある。前者は後者に比べ精度は良いものの解析時間が非常に長い。本研究では、flow-insensitive な別名情報と制御フローグラフの再調査によって別名が影響する箇所を調べ、その情報を元により多くの値をレジスタに置き換えることを目指した。実装は COINS (並列化コンパイラ向け共通インフラストラクチャ) 上で LIR （低水準中間表現）の最適化部として行った。SPEC CPU2000 のベンチマークプログラムで実験を行った結果、幾つかのプログラムで、flow-insensitive な別名情報のみを使ったレジスタ置き換えに比べて平均 3 %、最大 15 % の実行時間改善が見られた。

1　はじめに

一般的なコンパイラのバックエンドでは、別名と関わりのないローカル変数をレジスタに乗せるレジスタアロケーションという処理が組み込まれている。別名解析によって別名情報が得られたら、その結果をもとに部分的ながら、さらに多くの値をレジスタに乗せることが期待できる。

別名解析には大きく分けて flow-sensitive な解析 [3] と flow-insensitive な解析 [6] がある。前者は高精度な結果が得られるが、P-SPACE 問題であるため [2] 解析時間が長く、具体的には入力プログラムのサイズ n に対して、$O(n^4)$ 以上である [3]。それゆえ、場合によっては解析時間がかかり過ぎる恐れがある。一方後者は、精度が低いものの解析時間は前者に比べて大幅に短く、n に対する線形時間で解析を終えるものもある。しかし flow-insensitive な情報を元にした場合、その精度の低さゆえにレジスタに乗せる値をあまり増やせないと考えられる。

本研究では、flow-insensitive な別名情報を使い、さらにフローグラフの別名参照を再調査することで、より高い精度の情報を得る。そして、それを元により多くのメモリ参照をレジスタ参照に置き換えることを目指す。なお、置き換えの対象とするのはスカラー変数のみである。

本論文の構成は以下の通りである。第 2 節では、別名解析と、そのレジスタ参照への置き換えの関係について解説する。第 3 節では、本研究での前提と目標について述べる。第 4 節では本研究で行った内容とそのアルゴリズムについて述べる。第 5 節では、実装評価を行った環境や前提について述べる。第 6 節では実験結果を述べる。第 7 節では、まとめと今後の課題を述べる。

2　別名とレジスタ参照への置き換え

2.1　別名と別名解析について

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が存在する可能性がある。たとえば、ポインタ `p` がスカラー `x` を指しているとき、`*p` と `x` は別名である。別名が存在すると、メモリに間接的にアクセスされる可能性が出てくる。たとえば、変数 `x` のメモリ位置にアクセスする場合、`x=x+1` とするのではなく `x=*p+1` などとした場合は間接的なアクセス `*p` が存在することになる。

別名解析とは、プログラム中で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。例えば先ほどの例では、ポインタ `p` がスカラー `x` を指していたので、$\mathtt{p} \rightarrow \{\mathtt{x}\}$ という解析結果が得られる。

別名解析の分類の仕方には何通りかの方法がある

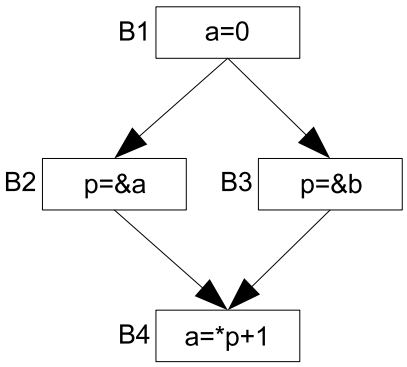

図 1　2 種類の別名解析

が、その中の 1 つとして、flow-sensitive なものと flow-insensitive なものの 2 種類に分ける方法がある。

flow-sensitive な別名解析はプログラムのフロー（流れ）を考慮した解析である。例えば図 1 のようなフローグラフで flow-insensitive な別名解析を行うと、次のような結果が得られる。

B1: p → {}
B2: p → {a}
B3: p → {b}
B4: p → {a,b}

このフローグラフには、パスの分岐があるが、それぞれのブロック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。例えばブロック B2 と B3 はプログラムの流れとしては分離しており、違うフローの中にある。そして、B2、B3 それぞれでポインタ p が参照しているものも異なる。

一方、flow-insensitive な解析は 1 つの手続き全体で共通の結果を出すためパスの流れを考慮したものにはなっておらず、それゆえ flow-sensitive なものよりも精度は低い。しかし、解析時間が flow-sensitinve なものに比べて短いという利点がある。アルゴリズムによっては、入力プログラムのサイズに対してほぼ線形時間で解析を終えるもの [8] もあり、本研究の実装にもそれを使っている。図 1 のフローグラフで flow-insensitive な解析をすると次のような結果が得られる。

p → {a,b}

このように、分岐などは考慮されず、あくまでもフローグラフ全体で 1 つの結果が得られている。

2.2　別名とレジスタ置き換えの関係

メモリ参照をレジスタ参照に置き換える際、別名の存在がネックとなる。以下にその例を示す。(A)(B) はそれぞれ目的コードをソースプログラム風に示したものである。

```
p=&a;                p=&a;
a=a+1;               ra=ra+1;
*p=1;                *p=1;

 (A)                  (B)
```

(A) のようなプログラムについて a をレジスタ参照に置き換えるとする。別名の存在を考慮せずに置き換えると、結果は (B) のようになる。ra はレジスタ参照を表している。

この例においては、a をレジスタ参照に置き換えると正しいプログラムにはならない。なぜなら*p が a の値がしまわれていたメモリ位置を参照するのに対して、ra はレジスタを参照しているからである。本来同一の値を更新しなければならない a=a+1 と*p=1 が、別々のものを更新するようになってしまうというわけである。(A) のように間接的にメモリの値を書き換えうるものが含まれている場合はレジスタ参照への置き換えが出来ない。

また、グローバル変数に関しては、手続き呼び出しが別名参照と同様の働きをする可能性がある。以下の例では、main から foo を呼び出しており、また、どちらの手続きでもグローバル変数 gx を参照している。

```
int gx=0;
main(){              foo(){
  gx=gx+1;             gx=gx+1;
  foo();             }
}
```

main の立場で見れば、foo() というものを呼び出すことで、直接 gx にアクセスすることなく gx を参照するので、foo() は gx の別名を参照するものとして扱わなくてはならない。foo の中身を考慮することなく main の gx をレジスタ参照に置き換えてはならない。

3　研究の目標

3.1　仮想レジスタについて

プログラム中の値で、レジスタに乗せられる候補が多数あったとしても、実際にそれらすべてをレジスタに乗せられるとは限らない。なぜなら、レジスタの数には限

$$AliasSource(B_i) = \{v_j \mid Alias\ reference\ by\ source\ v_j\ is\ in\ block\ B_i\}$$
$$Target(v_i) = \{v_j \mid v_j\ is\ alias\ target\ of\ v_i\}$$
$$AliasedBlock(v_i) = \{B_j \mid AliasSourse(B_j)\ contains\ \ v_k\ \ such\ that\ Target(v_k)\ contains\ \ v_i\}$$

表 1　AliasedBlock 等の定義

りがあるからである。大抵の場合はレジスタの数は 6〜32 本程度である（多いものでは 64 本などもある）。そこで、どの値を実際にレジスタに乗せるのか選別が行われる。さて、選別の候補となる値に関しては、あらかじめ候補であることをはっきりさせておきたい。そこで、コンパイルの流れの中で、レジスタアロケーションが行われる前のフェーズでは、レジスタに乗せる候補の値を**仮想レジスタ**というもので置き換えておく。

この仮想レジスタというものは、中間言語において用いられる概念であり、仮想レジスタに置き換えられた値が実際にレジスタに乗るかどうかはレジスタアロケーションによって決まる。

3.2　研究の目標

本研究の目的は、flow-insensitive な別名解析結果に次節以降で説明する手法で情報を付加することで、より精度の高い情報を得て、その結果を元により多くの値を仮想レジスタに置き換えることである。

より多くの値を仮想レジスタに置き換えておけば、より多くの値がレジスタに乗ると思われる。すなわち、メモリ参照がレジスタ参照に置き換えられるのでプログラムの実行時間改善が期待できる。

また、本研究で提案する手法を使うことによるコンパイル時間の増大にも留意したい。flow-insensitive な別名解析の長所は、解析時間が短いという点である。本研究では flow-insensitive な別名情報を補うという作業を行うわけだが、この作業に長い時間をかけて別名情報を補ってもあまり意味がない。flow-insensitive な別名解析の長所を打ち消すことのないよう、ある程度短い時間で作業を終える必要がある。

4　アルゴリズム

4.1　手法 A(ブロックごとの別名参照チェック)

flow-insensitive な解析では、１つのフローグラフについて、そのフローグラフ全体で共通の別名情報が得られる。例えばフローグラフ A について `p` → `{a,b}`という解析結果が出たら、フローグラフのどの部分でも `p` は `a` か `b` を指していると考えなくてはならない。よって、A 全体で `a`、`b` は仮想レジスタに乗せることは出来ないと判断せざるを得なくなる。

しかし、実際にレジスタ参照への置き換えを妨げるのは `p` が `a`、`b` を指しうるという事実ではなく、`*p` によって `a`、`b` の値が別名参照されているということである。つまり、実際に別名参照が行われている部分で値をメモリにしまっておけば、それ以外の場所では値を仮想レジスタに乗せることが出来る。

本研究では、このことを基本ブロックを単位として解析した。フローグラフ中の各変数 v について、v が別名参照を受けていないブロックでは、v を仮想レジスタに乗せる。v が実際に別名参照を受けているブロック B については以下の 2 つの処理を行う。なお rv は仮想レジスタを表している。

- B の前に入口ブロックを挿入する。入口ブロックには v の値を仮想レジスタからメモリに戻す命令 $v = rv$ をセットする。
- B の後に出口ブロックを挿入する。出口ブロックには v の値をメモリから再び仮想レジスタに乗せる命令 $rv = v$ をセットする。

この 2 つの処理によって v が別名参照を受けるブロックでは、v の値をメモリにしまっておくことが出来る。上述の手法を本論文では便宜上「手法 A」と呼ぶ。

手法 A の例を図 2 に示す。別名解析の結果は $p \rightarrow \{a, b\}$ とする。この例で、a、b を別名参照しているのはブロック B1 のみである。そこで、ブロック B1 の前に B1 の入口ブロック B3 を挿入し、a と b の値をメモリにしまう。また、ブロック B2 の後に B2 の出口ブロック B4 を挿入し、a と b の値をレジスタに乗せる。

以下、手法 A の説明を行う。手法 A では、あらかじめ flow-insensitive な別名情報が得られていることが前提となる。

この手法において、フローグラフ中の各変数 v_i について別名参照を受けるブロックの集合 $AliasedBlock(v_i)$ を求めなくてはならない。フローグラフ中に存在するブロックの集合 B を $B = \{B_1, B_2, \ldots, B_w\}$、フローグラフ中の変数の集合 V を $V = \{v_1, v_2, \ldots, v_m\}$ とするとき、$AliasedBlock(v_i)$ を求める式は表 1 のようになる。

各式の内容は次のようになっている。

- $AliasSourse(B_i)$: ブロック B_i に含まれる別名参照の参照元の集合。フローグラフを再調査することで求める。(例えば*p があれば、その参照元は p)
- $Target(v_i)$: 変数 v_i が指す変数の集合。あらかじめ求まっている flow-insensitive な別名情報を参照すれば求められる。
- $AliasedBlock(v_i)$: 変数 v_i が別名参照を受けるブロックの集合。

フローグラフ中の各変数 v について $AliasedBlock(v)$ を求めたら、仮想レジスタへの置き換えの準備が整うことになる。

また、手法 A 全体のアルゴリズムは**アルゴリズム**1 のようになる。checkAlias で、フローグラフ中の各変数 v について $AliasedBlock(v)$ を求め、convert で仮想レジスタへの置き換えと、別名参照を受けるブロックについてのメモリ-レジスタ転送命令の挿入を行っている。

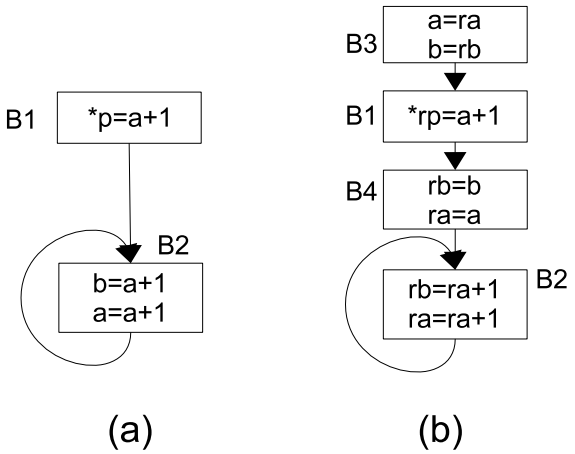

図 2　手法 A(ブロックごとの別名参照チェック) の例

手法 A のアルゴリズムを再度、例を用いて示す。図 2 の 2 つのフローグラフは左が置き換え前、右が置き換え後のものである。まず、与えられた別名情報は次の通りであるとする。

Alias Information			
Source	a	b	p
Target	-	-	a,b

アルゴリズム 1 にしたがって、フローグラフ中の各変数について $AliasedBlock$ を求めていく。

まずブロック B1 には、別名参照*p が含まれている。この参照元は p である。与えられている別名情報より、p は a または b を指すことが分かるので、ブロック B1 では、a、b が別名参照を受けているということになる。よって、

$$AliasedBlock(a) = B1 \cup AliasedBlock(a)$$
$$AliasedBlock(b) = B1 \cup AliasedBlock(b)$$

とすることで、$AliasedBlock(a)$、$AliasedBlock(b)$ それぞれに B1 を追加する。

ブロック B2 には別名参照がないので $AliasedBlock$ へのブロックの追加は必要ない。

以上より、フローグラフに関して次の情報が得られる。この表は、フローグラフ中の各変数がどのブロックで別名参照を受けているかを示したものである。

Flow Graph Information			
Variable	a	b	p
AliasedBlock	B1	B1	-

この結果より、変数 a,b はブロック B1 で参照を受けることが分かる。p が別名参照を受けるブロックはこのフローグラフにはない。

よって、a については次のようになる。まず、B2 は AliasedBlock(a) に含まれていないので、B2 では a を仮想レジスタ ra に置き換える。一方、B1 は AliasedBlock(a) に含まれる、つまり a は B1 で別名参照を受けるので、B1 について次の処理を行う。

B1 の入口ブロック B3 を作り、B3 に命令 $a = ra$ をセットする。
B1 の出口ブロック B4 を作り、B4 に命令 $ra = a$ をセットする。

これによって、B1 では a の値がメモリにしまわれた状態になる。

b、p についても同様の処理を行う。その結果、図**??**の右側のフローグラフのようになり、手法 A の適用は終了する。

4.2　手法 B(ブロックのグループ化)

ある変数に対して、別名参照を含むブロックが連続して続いていた場合、ブロックごとの別名チェックのみでは、ブロック間に無駄なメモリ-レジスタ転送命令を挿入することになってしまう。例えば、図 3 において、与えられた別名情報が $p \rightarrow \{a\}$ であるとする。フローグラフ (a) に対して手法 A を適用するとフローグラフ (b) のようになる。

(b) のブロック B5、B6 に注目すると、この 2 つのブ

アルゴリズム 1 手法 A のアルゴリズム

algorithm for A()

1: $checkAlias()$
2: $convert()$

checkAlias()

3: **for all** $v_k \in V$ **do**
4: $\quad AliasedBlock(v_k) = \phi$
5: **for all** $B_i \in B$ **do**
6: $\quad$ **for all** $v_j \in AliasSource(B_i)$ **do**
7: $\quad\quad$ **for all** $v_k \in Target(v_j)$ **do**
8: $\quad\quad\quad AliasedBlock(v_k) = B_i \cup AliasedBlock(v_k)$

convert()

9: **for all** $B_i \in B$ **do**
10: $\quad$ **for all** $v_j \in V$ **do**
11: $\quad\quad$ **if** $AliasedBlock(v_j)$ contains B_i **then**
12:
13: $\quad\quad\quad$ Insert memory-register move instructions
14: $\quad\quad$ **else**
15: $\quad\quad\quad$ Convert v_j to virtual register

ロックで行われていることは、a の値をメモリから仮想レジスタに乗せ、またメモリに戻すということのみである。よって、B5、B6 は削除しても差し支えない。

このように、同じ変数が別名参照を受けているブロックが有向辺でつながっている場合、それらのブロック一つ一つに対してメモリ-レジスタ転送命令を挿入すると、ブロック間に挿入されたメモリ-レジスタ転送命令は無駄なものとなる。

そこで、各変数に関して、別名参照のあるブロック同士が直接有向辺でつながっている場合は、そのブロック同士をグループとしてまとめることにする。そして、グループ外からグループ内への入口でレジスタの値をメモリに戻す命令を、グループ内からグループ外への出口でメモリの値をレジスタに移す命令を挿入すればよい。

この手法を、本論文では便宜上「手法 B」と呼ぶ。手法 B は、ブロックのグループ化を取り入れることで手法 A を拡張したものである。

ブロックのグループ化は、各変数 v が別名参照を受けるブロックの集合 $AliasedBlock(v)$ の要素ブロック同士で行うことになる。グループ化を行うと、いずれかのグループのメンバーになるブロックと、グループを形成せず単独のままのブロックに分かれる。よって以下の 2 つ

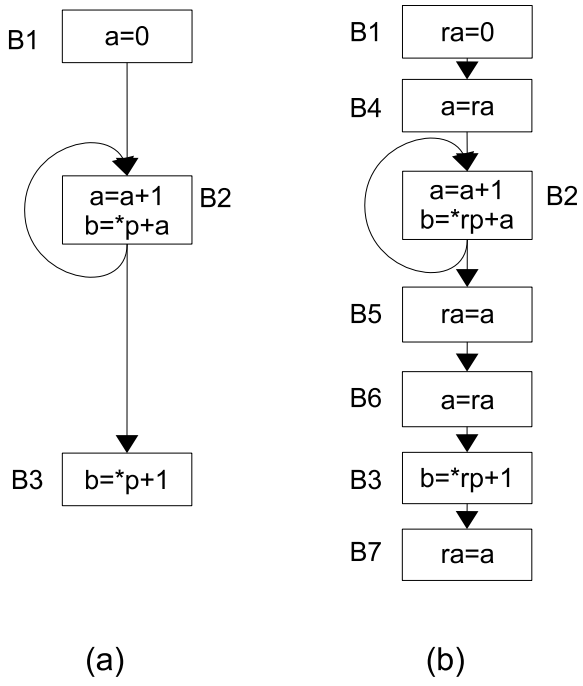

図 3　ブロックのグループ化が有効な例

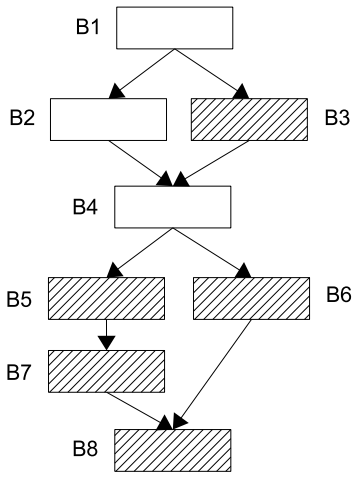

図 4　ブロックのグループ化の例

の集合を定義する。

- $groupList(v)$：グループ化によって出来たグループの集合。
- $singleList(v)$：単独のままのブロックの集合。

$AliasedBlock(v)$ の要素ブロックは、グループとして $groupList(v)$ に、または単独のまま $singleList(v)$ に振り分けられることになる。

$AliasedBlock(v)$ が与えられたとき、その要素を $groupList(v)$ と $singleList(v)$ に振り分けるアルゴリズムは、**アルゴリズム**2 のようになる。getMember(BasicBlock b) により、$AliasedBlock(v)$ に含まれるブロックのうち、ブロック b と有向辺でつながっているブロックを再帰的にたどり、グループを形成する。connected(BasicBlock b1, BasicBlock b2) はブロック b1 とブロック b2 が有向辺で結ばれていれば true を返す。なお、このアルゴリズムは、$AliasedBlock(v)$ の要素を $groupList(v)$ と $singleList(v)$ に振り分ける際、振り分けられたブロックは $AliasedBlock(v)$ から削除されていき、最後は $AliasedBlock(v)$ は空になる。手法 B においては、$groupList(v)$ と $singleList(v)$ さえ得られていれば仮想レジスタへの置き換えが出来るので、$AliasedBlock(v)$ は空になっても差し支えない。

ブロックの形成について例を示しておく。図 4 のフローグラフにおいて、斜線をつけたブロックは、フローグラフ中の変数 v が別名参照を受けるブロックであるとする。つまり、$AliasedBlock(v) = \{B3, B4, B5, B7, B8\}$ である。

B4 と B8 が有向辺でつながっているため、グループ $\{B4, B8\}$ が形成される。しかし、B8 と B7 も有向辺でつながっているため、B7 も B8 と同じグループに属することになる。よって、グループ $\{B4, B8\}$ に B7 が加わり $\{B4, B7, B8\}$ となる。B5 と B7 も有向辺でつながっているので、B5 も加わり、最終的にグループ $\{B4, B5, B7, B8\}$ が出来上がる。B3 については、他の斜線のついたブロックとつながっていないので単独のまま残る。

手法 B を少し複雑な例を用いて示す。図 5 の 2 つのフローグラフは左が置き換え前、右が置き換え後のものである。まず、与えられた別名情報は次の通りであるとする。

Alias Information				
Source	a	b	p	q
Target	-	-	a,b	a

また、手法 1 で述べた方法により、フローグラフの情報は次のようになる。

Flow Graph Information				
Variable	a	b	p	q
AliasedBlock	B1,B2,B3	B1,B3	-	-

この情報により、各変数について、別名参照を受けているブロックのグループ化を行う。

まず a について、別名参照を受けているブロックは B1、B2、B3、つまり $AliasedBlock(a) = \{B1, B2, B3\}$ である。

$$connected(B1, B2) = true$$
$$connected(B1, B3) = true$$

より、アルゴリズム 2 に従うと、

$$groupList(a) = \{B1, B2, B3\}$$
$$singleList(a) = \phi$$

となる。他の変数についても同様にして $groupList$ と $singleList$ を求める。その結果、次のようになる。

Flow Graph Information				
Variable	a	b	p	q
groupList	B1,B2,B3	B1,B3	-	-
singleList	-	-	-	-

a について、グループは $\{B1, B2, B3\}$ であり、グループに属さないブロックはない。よってグループ $\{B1, B2, B3\}$ に関してメモリ-レジスタ転送命令の挿入を次のように行う。

アルゴリズム 2 手法 B におけるグループ化のアルゴリズム

algorithm for B()

1: **while** $AliasedBlock(v_i) \neq \phi$ **do**
2: $tempList = \phi$
3: $b = the\ first\ element\ of\ AliasedBlock(v_i)$
4: $tempList = b \cup templist$
5: $remove\ b\ from\ AliasedBlock(v_i)$
6: $getMember(b)$
7: **if** size of $tempList$ is 1 **then**
8: $singleList(v_i) = b \cup singleList$
9: **else**
10: $groupList(v_i) = tempList \cup groupList$

getMember(BasicBlock b)

11: **for all** $B_j \in AliasedBlock(v_i)$ **do**
12: **if** $connected(b, B_j) =$ **true** $\wedge\ B_j \notin tempList$ **then**
13: $tempList = B_j \cup tempList$
14: $remove\ b\ from\ AliasedBlock(v_i)$
15: $getMember(B_j)$

connected(BasicBlock b1, BasicBlock b2)

16: **if** b1 and b2 are connected with an edge **then**
17: **return true**
18: **else**
19: **return false**

- グループの入口ブロック B5 を作り、命令 $a = ra$ を挿入する。
- グループの出口ブロック B7、B8 を作り、命令 $ra = a$ を挿入する。

グループにも $singleList(a)$ にも属さないブロックでは、a を ra に置き換える。

他の変数についても同様のことを行う。すべての変数について仮想レジスタへの置き換えと、メモリ命令の挿入を行うと図 5 の右側のフローグラフのようになる。このように、ブロックのグループ化は、基本的には手法 A を適用した際に生じる無駄なメモリ命令を削除した結果になる。

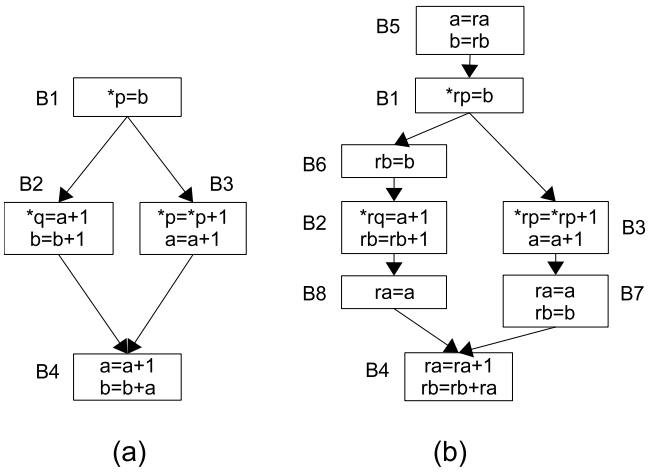

図 5　手法 B(ブロックのグループ化) の例

4.3 レジスタへの置き換えによる利得

前節で提案した二つの手法は、別名参照が行われているブロック、またはグループに関してメモリ命令を挿入しなくてはならない。これによって増大したメモリ命令が、仮想レジスタ割り当ての効果を打ち消してしまうことがある。図 6 がその例である。なお、ここでいうメモリ命令とは、メモリ-レジスタ転送命令を意味する。

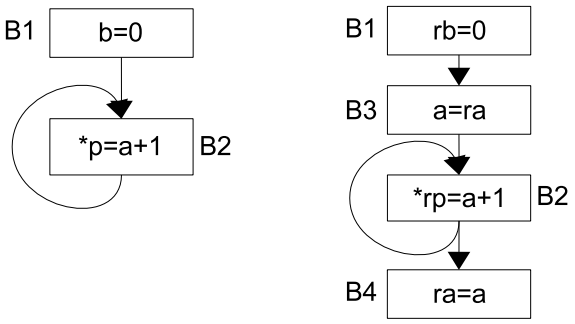

図 6　逆効果となる例

変数 a を仮想レジスタに割り当てる作業を行ったものの、仮想レジスタ割り当てによって削減されたメモリ命令数が、挿入されたメモリ命令数を下回っているため、結果としてはメモリ命令が増えることになってしまう。つまり、メモリアクセスの増大であり、実行時間が長くなる可能性が高い。

もし、このフローグラフに対応する手続きが頻繁に呼び出されるとすれば、事は深刻になる。例えば手続き内でのメモリ命令数の増加が 1 であったとしても、手続きが 100 回呼び出されると、通常より 100 回多くメモリ命令が実行されることになる。グローバル変数の仮想レジスタ割り当てに関しては手続き呼び出しも別名参照とみなすので、ローカル変数の場合に比べてメモリ命令の挿入数も多くなると考えられる。

このメモリ命令の増大による損失を防ぐために、利得 (Benefit) の計算を導入する。つまり、効果が見込めるスカラー変数のみを、本研究の仮想レジスタ割り当ての対象とするわけである。

変数 v について、v を仮想レジスタに置き換え可能な箇所数を $Convert_v$ とする。その 1 箇所につき、仮想レジスタに置き換えることで v のロード命令 1 つとストア命令 1 つが削減できることになる。よって、削減できるメモリ命令数は $2 \times Convert_v$ となる。ただし、次のような例で、(A) のようなものに対して a を仮想レジスタに乗せると (B) のようになる。a を 2 箇所で仮想レジスタに置き換えているが、削除されたメモリレジスタ転送命令は a=a+1 の前後にあった 2 つだけである。このように、計算にずれが生じる場合があるが、$2 \times Convert_v$ として近似した。

```
a=a+1;                ra=ra+1;
 (A)                   (B)
```

x が別名参照を受けているブロックやグループについて新しく挿入するメモリ命令数を $Insert_v$ とする。変数 x の仮想レジスタ置き換えによる利得 $Benefit_v$ を

$$Benefit_v = 2 \times Convert_v - Insert_v$$

と定義する。$2 \times Convert_v$ は v を仮想レジスタに置き換えることで削減できるメモリ命令数、$Insert_v$ は v を仮想レジスタに置き換えることで挿入が必要になるメモリ命令数であるので、$Benefit_v$ は、v を本論文で提案した手法によって置き換えたときに全体として削減できるメモリ命令数である。$Benefit_v$ が 0 を超えたら、v は仮想レジスタへの置き換えが有効であるとみなし、置き換えを行う。

一般的に、ロード命令の方がストア命令に比べて処理時間がかかるが、この違いは無視する。また動的なメモリ命令等の回数を考慮した場合、単純にこの計算方法が効果的であるとは限らないが、ある程度の効果は期待できるので一つの近似的方法としてこの方法を取り入れることにする。

5　実装

5.1　COINS

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テーマに 2000 年より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。

COINS は、高水準共通中間表現 HIR(high-level intermediate representation) と、低水準共通中間表現 LIR(low-level intermediate representatoin) を持ち、ソース言語から HIR への変換部、HIR での各種最適化部、HIR から LIR への変換部、LIR での各種最適化部、LIR から機械語への変換部を組み合わせることによってコンパイラを実現している。COINS の概念図を図 7 に示す。

5.2　実装環境と前提

実装には COINS [4] のバージョン 1.3.2.3-ja を採用し、LIR（低水準中間表現）の最適化部の最適化の一つとして、前節で提案した手法 A、B を実装した。

レジスタ参照への置き換え対象はグローバル、ローカルのスカラー変数のみとする。また、対象言語は C 言語で、シングルスレッドのプログラムのみを対象とする。

また、プログラム中での別名参照の連鎖レベルは 2 重までとする。つまり***p のようなものは存在しないことを前提とする。これ以上の多重連鎖の場合を扱わなくても、単に解析が保守的になるだけであり、正しさ自体は保たれる。また実用の面から考えてもこの程度でよいと判断して差し支えない。

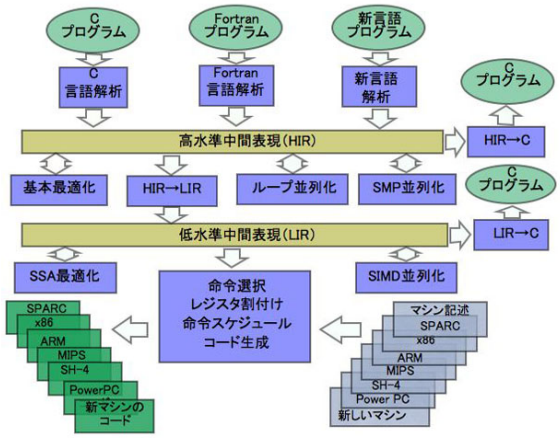

図 7　並列化コンパイラ向け共通インフラストラクチャ COINS 概念図

5.3　採用した別名解析

本研究で使う flow-insensitive な別名情報を得るために、Steensgaard の別名解析 [8] を採用した。

Steensgaard の別名解析アルゴリズムの特徴は、変数のメモリ上の位置を記述するために型と呼ばれるものを導入し、型に関する推論規則を用いてポインタ解析を行うという点である。

また、スカラーだけでなく配列や、構造体なども一つの変数とみなされる。例えば、配列 a とスカラー b に対して a[1]=&b とすれば、この式に関する解析結果は a → {b}となる。

さらに、入力プログラムのサイズ N に対して、解析時間は、$O(N\alpha(N,N))$ である（α はアッカーマン関数の逆関数)。これはほぼ N の線形時間に等しい。

6　実験結果と考察

本節では、前節で提案した手法の実験結果とそれに対する考察を行う。

6.1　実験環境と内容

実験は、Sun Microsystems の Sun Blade 1000 で行った。この主な仕様は、表 2 の通り。

テストプログラムには SPEC CPU2000 のベンチマークプログラムより、

164.gzip, 175.vpr, 181.mcf, 197.parser, 254.gap, 256.bzip2, 300.twolf, 183.equake, 188.ammp, 179.art

Architecture	Superscalar SPARC Version 9
CPU	750MHz UltraSPARCIII × 2
Cache	64KB データ 32KB インストラクション
Memory	1GByte
OS	SunOS 5.8

表 2　Sun Blade 1000 の主な仕様

の 10 個を用いた。

実験は、以下に述べる 5 種類を行った。すべての実験は、COINS に既に実装されている標準的な SSA 最適化を適用してコンパイルを行い、それぞれ 3 回ずつ実行し実行時間の中央値を取った。なお、上述の標準的な最適化は以下の通りである。

- prun: Pruned SSA 形式への変換
- cpyp: 条件分岐を考慮したコピー伝播
- cse: 共通部分式削除
- cstp: 定数畳み込みと定数伝播
- dce: 無用命令削除
- divex: 式を 3 アドレス方式に変換（代入の右辺の演算子はたかだか 1 つ）
- hli: ループ不変コードの巻き上げ
- osr: ループの帰納変数に関わる演算の強さの軽減と判定の置き換え
- preqp: 質問伝播の方法による大域的値番号付けと部分冗長性除去
- rpe: 冗長な Phi 関数の除去
- srd3: Sreedhar の方法 III による SSA 逆変換

これらは SSA 形式で適用する最適化であり、

prun/divex/cse/cstp/hli/osr/hli/
cstp/cpyp/preqp/cstp/rpe/dce/srd3

の順で適用した。この順で使用する最適化セットを、便宜上「最適化セット A」と呼ぶ。実験 5 種類の内容は次の通りである。

実験 1 (coins)

COINS デフォルト + 最適化セット A で、各ベンチマークプログラムに対するコンパイル時間、実行時間の計測を行った。COINS デフォルトとは、COINS で最適化オプションを用いないという意味である。ただし COINS では、最適化オプションなしでも何種類かの簡

単な最適化は自動的にかかるようになっている。

実験２ (regpro)

COINS デフォルト + 最適化セット A+ グローバル変数のレジスタプロモーション [9] で、各ベンチマークプログラムに対するコンパイル時間、実行時間の計測を行った。このレジスタプロモーションは

- 別名参照を含まない
- 手続き呼び出しを含まない

の２条件を満たす自然ループ内で一時的にグローバル変数の値のみをレジスタに乗せるというかなり限定されたものである。

実験３ (steens)

COINS デフォルト + 最適化セット A+Steensgaard の別名解析結果のみを使った仮想レジスタ割り当て(本研究で提案した手法は使わない) で、各ベンチマークプログラムに対するコンパイル時間、実行時間の計測を行った。

「SteensGaard の別名解析結果のみを使った仮想レジスタ割り当て」は、Steensgaard の別名解析の結果アドレスを取られていない (どの変数にも指されていない) ローカル、グローバルのスカラー変数をフローグラフ全体で仮想レジスタに置き換えるというものである。ただし、手続き呼び出しのあるフローグラフでは、グローバル変数は置き換えない。

実験４ (bcheck)

COINS デフォルト + 最適化セット A+ 本研究の手法 A(Steensgaard の別名解析結果を使う) で、各ベンチマークプログラムに対するコンパイル時間、実行時間の計測を行った。

実験５ (bgroup)

COINS デフォルト + 最適化セット A+ 本研究の手法 B(Steensgaard の別名解析結果を使う) で、各ベンチマークプログラムに対するコンパイル時間、実行時間の計測を行った。

6.2 実験結果と考察

実行時間の計測結果を図 8 のグラフに示した。表 3 には、実行時間を数値として示した。また、全コンパイル時間の計測結果を図 9 のグラフに示した。各バーの内訳は次のようになっている。

164.gzip、175.vpr、181.mcf、197.parser の４つについては、実験３の結果が出ていない。これは、この４つのプログラムでコンパイルエラーが起き、コンパイルを完了できなかったためである。

	coins	regpro	steens	bcheck	bgroup
164.gzip	806	805	-	758	757
175.vpr	775	776	-	767	765
181.mcf	467	470	-	498	495
197.parser	872	869	-	862	862
254.gap	701	703	694	687	688
256.bzip2	882	880	888	853	853
300.twolf	1116	1194	1054	1103	1105
183.equake	974	974	975	975	976
188.ammp	1431	1428	1412	1419	1415
179.art	626	627	642	533	534

表 3　実行時間　単位は (秒)

本研究の手法の効果は steens と bcheck、bgroup の比較で分かる。全体的に本研究で提案した手法の効果は顕著に現れたといえる。ベンチマークプログラム内で、別名参照を受けている値は、別名参照を受けていない値に比べて極端に少なかった。しかし、254.gap、256.bzip2 において、bcheck、bgroup 共に、steens に比べて数 %、179.art では 15 % もの実行時間の改善が見られた。164.gzip、175.vpr、197.parser の３つでは steens の結果が出ていないが、他のベンチマークプログラムでの傾向からして、数 % の効果があるものと予想できる。

bcheck、bgroup は、181.mcf を除くどのベンチマークプログラムについても coins よりも良い結果を出しているが、これはレジスタ置き換えによる利得の計算が一定水準以上の結果を保っているためと言えるだろう。

また、181.mcf を除くすべてのベンチマークプログラムで bcheck、bgroup の実行時間は regpro 以下に抑えられている。本研究の手法は、過去の研究で実装したレジスタプロモーションに比べて実行時間改善効果が大きいことがわかる。しかし、このレジスタプロモーションは、グローバル変数のみを対象としており、別名参照や手続き呼び出しのないループにおいてのみ値をレジスタに乗せるという保守的なものであるので、本研究の手法のほうが良い結果を出すのは必然と言えるかもしれない。

300.twolf、188.ammp については、steens に比べて実行時間が長くなったが、これには２つの理由が考えら

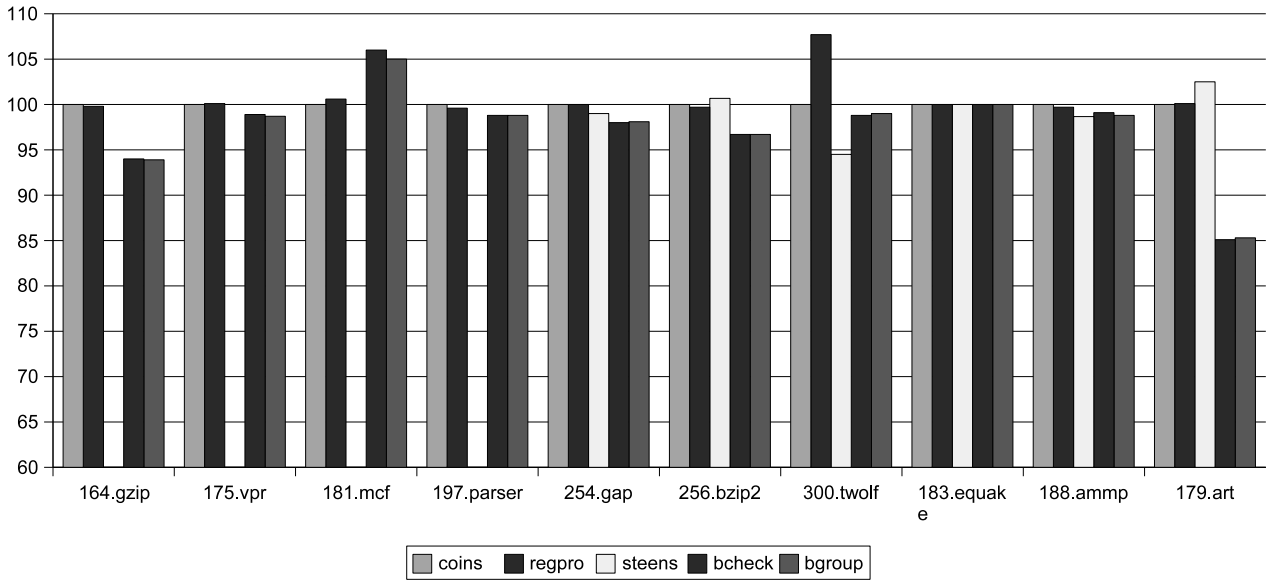

図 8　SPEC のベンチマークプログラムの実行時間 (coins に対する相対比 (%))

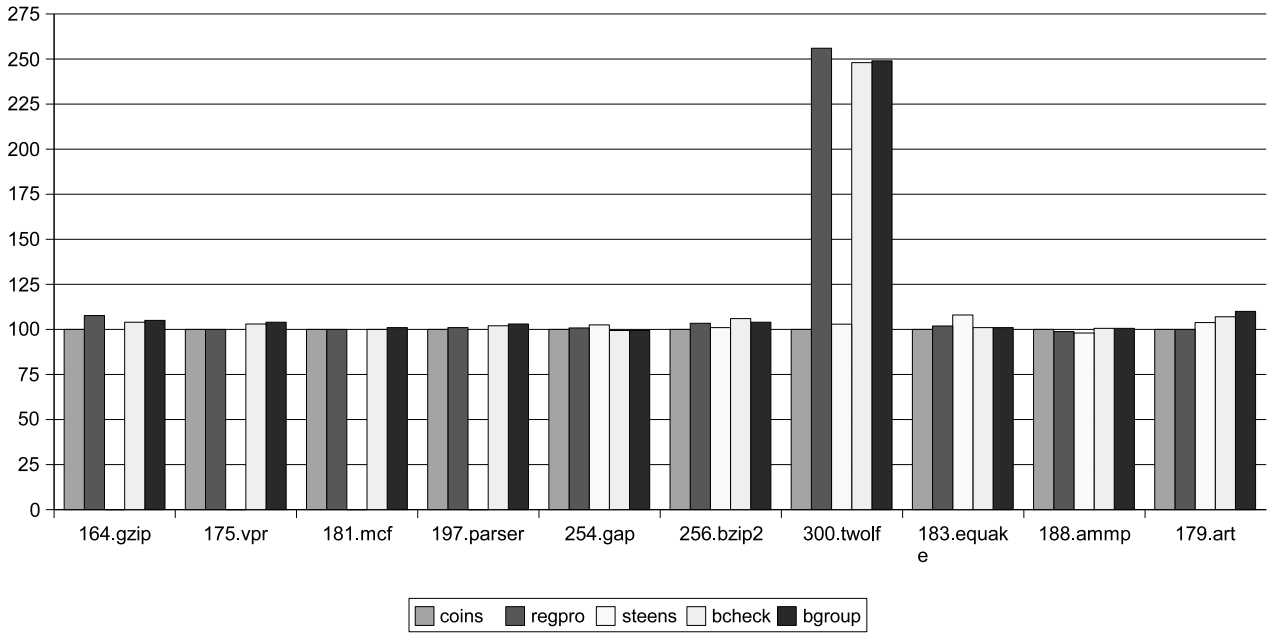

図 9　SPEC のベンチマークプログラムのコンパイル時間 (coins に対する相対比 (%))

れる。

1 つ目は、レジスタ圧力の増大である。グローバル変数のレジスタプロモーション [9] においても、レジスタ圧力の増大により 300.twolf の実行時間は長くなるという結果が出ている。

2 つ目は、レジスタ置き換えによる利得の計算に起因するものである。利得の計算においては、レジスタへの置き換え箇所数と、メモリ命令の挿入個数を静的にカウントしている。そのため、利得計算の結果ではレジスタへの置き換えが有効であるとされても、動的にはレジスタへの置き換えがメモリ命令数を増大させるということが考えられる。

181.mcf についても、傾向としては本研究で提案した手法を使った場合の実行時間が増大している。181.mcf は他のベンチマークプログラムと比べて小さめのプログラムなので、レジスタ圧力が実行時間の増大を引き起

こしたとは考えにくい。利得の計算の大雑把さが他のベンチマークプログラムより悪い形で表れた結果だと思われる。

実行時間に変化のなかった 183.equake については、別名参照を受けているもの自体がごく少数であった、候補の変数のほとんどがレジスタ置き換えによる利得がマイナスであった、などの理由が考えられる。

本研究で提案した、手法 A と手法 B の比較についても述べる。全体的に bcheck と bgroup では特記するほどの実行時間の差はないため、ブロックのグループ化の効果は大きくないといえる。僅かな差ながら、164.gzip、175.vpr、181.mcf、188.ammp では bgroup が bcheck よりも実行時間が短い。また、300.twolf、179.art では bcheck よりも bgroup の方が実行時間が長い。手法 B によって、手法 A に比べてメモリ命令が減るはずである。300.twolf、179.art の 2 つでの実行時間増大は、誤差と捉えるべきか否か微妙なレベルのものである。これを誤差でないとする場合は利得計算が原因であると考えられる。

ブロックのグループ化によってメモリ命令が省かれるということは、bgroup では利得計算の結果が 0 を超える変数が bcheck に比べて増える可能性があるということである。つまり、bgroup ではより多くの変数が仮想レジスタに乗っていると考えられる。

利得の計算に間違いが生じうることについては既に述べた。bcheck の場合と比べて多くなった分の変数について、実は利得がマイナスにもかかわらず利得計算の結果では 0 を超える場合、これらを仮想レジスタに乗せることで bcheck にはなかった損失が生まれることになる。この損失が大きければ、bgroup の実行時間が bcheck よりも実行時間が遅くなることも考えられる。

本研究で提案した手法を使ったコンパイル時間 bcheck、bgroup はほとんどのベンチマークプログラムについて、coins と比べた場合の増大を小さく抑えることができた。中には coins のコンパイル時間に比べて bcheck、bgroup のコンパイル時間がわずかに短いものがある。理論的に考えて、coins よりも bcheck、bgroup が短くなるとは考えにくいので、これは誤差として考えるべきである。

300.twolf に関しては、bcheck、bgroup が coins の約 2.5 倍程度も増加している。他のベンチマークプログラムので比較を勘案すると、このコンパイル時間の増加の原因は本研究で提案したアルゴリズムの効率の悪さとは考えにくい。実行時間の考察でも触れたレジスタ圧力の増大によって、レジスタ割り当てフェーズでの時間が増えたとみるべきであろう。

7 まとめと今後の課題

本研究では、flow-insensitive な別名解析結果とフローグラフを再調査した結果を組み合わせ、それを元にスカラー変数をより多く仮想レジスタにのせる手法を提案した。さらに、その手法を COINS 上で実装し、実行時間、コンパイル時間の計測を行った。結果として、ほとんどの場合目的コードの実行時間を改善でき、さらにコンパイル時間の増大を小さく抑えることが出来た。ただし、本研究の手法を適用することでレジスタ圧力を上げ過ぎてしまうようなプログラムに関しては、実行時間、コンパイル時間ともに長くなってしまうようである。

今後の課題として、主に二つがあげられる。

１つ目は、本研究の手法によるレジスタ圧力の増大を防ぐことである。しかし、レジスタ圧力はコンパイル時に使う最適化等の関係から、正確に求めるのは困難であると思われる。現時点では、有用な手段のあてはない。

２つ目は、本研究の手法における利得の計算のアルゴリズムの改善である。本研究では、単にコードの字面上のカウントにより利得を計算した。ループ構造などによる命令の実行頻度の違いを考慮すれば、さらに正確な計算ができる可能性がある。例えば、フローグラフ中のすべてのループの実行回数を 10 回と仮定して、近似的に計算するなどの方法がある。また、静的なデータを元にするのではなく、プロファイルを取るなどの手段により得られる動的なデータを元にすれば、より信頼のおける利得計算が出来る可能性があり、それが実行時間を大きく改善することも十分ありうる。

謝辞

本研究の一部は、日本学術振興会科学研究費補助金の補助を受けた。また、別名解析の実装にあたって協力してくださった吉羽和之氏に感謝する。

参考文献

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison Wesley, 2003.

[2] Venkatesan T. Chakaravarthy. New results on the

computability and complexity of points–to analysis. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 115–125, New York, NY, USA, 2003. ACM Press.

[3] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 232–245, Charleston, South Carolina, 1993.

[4] COINS-Project. Coins-project homepage. http://www.coins-project.org/.

[5] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, 2001.

[6] Vishwanath Raman. Pointer analysis – a survey, 2004.

[7] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction*, pp. 136–150, London, UK, 1996. Springer-Verlag.

[8] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pp. 32–41, 1996.

[9] 狩野祐介. flow-insensitive な別名情報とフロー情報を用いた最適化対象の拡大. Master's thesis, 東京工業大学大学院 情報理工学研究科, 2007.