実行時情報を利用した部分冗長除去と SSA 形式への適用

伊藤 陽*　　佐々 政孝

2006 年 2 月 10 日

1 はじめに

部分冗長除去 (Partial Redundancy Elimination, PRE) [*1]は、共通部分式除去とループ不変式移動の効果を含んだ効果的な最適化である。PRE は、Morel らによって最初のアルゴリズムが提案され [14]、その後も多くの改良されたアルゴリズムが提案されている [5, 6, 7, 11, 12, 15]。

これらの研究の中で、PRE を**静的単一代入 (Static Single Assignment, SSA) 形式**[*2]上で行おうという試みがある。SSA 形式は最適化に適したプログラムの表現形式であり、近年盛んに研究が行われている。しかし、PRE を SSA 形式上で行おうとすると、

- 通常形式上では同一の変数であったものが名前替えにより異なる変数名になることがある。
- SSA 形式での変数には満たすべき条件があるので、異なるブロックにコードを移動する際に変数をそのまま移動できない。

といった困難がある。

ところで、PRE の基本的な手順は、

1. プログラムに式を挿入し、部分冗長な式を全冗長にする。
2. 全冗長となった式を除去する。

である。この、式の挿入点の違いがさまざまなアルゴリズムの差であり、最適化効果の差となる。多くの PRE アルゴリズムでは、計算量を増加させる可能性のある場所には式を挿入しないことにしている。これを**保守的な方法 (conservative method)** と呼ぶ。これに対して溝渕は、計算量を増加させる可能性があっても式の挿入を行う PRE アルゴリズムを提案している [13]。このような方法を**積極的な方法 (aggressive method)** と呼ぶ。また、Cai らは実行時情報を利用した方法を提案している [3]。

本研究では、次のような特徴をもった PRE アルゴリズムを提案する。

- SSA 形式のまま処理を行う一般性のある方法を示す。
- 実行時情報を利用することによって、保守的な方法と積極的な方法の長所を合わせたコード移動を行うことができる。
- コピー伝播を組み合わせることによって、今まで移動が困難であった式を移動させることができる。

また、この方法を **PBPRE (Profile-Based Partial Redudancy Elimination on SSA form)** と名づける。

本研究では Cai らのアルゴリズム [3] をもとに、実行時情報を利用した PRE のアルゴリズムを開発した。また、PRE アルゴリズムの中でコピー伝播を用いることによって、冗長な式を除去したことによって新たに生まれた冗長性を連鎖的に除去する手法を提案した。また、通常形式のアルゴリズムとして開発したものを変形することによって SSA 形式に対応させる一般性のある方法を考察した。この通常形式のアルゴリズムを SSA 形式のアルゴリズムに変形する手法は他の通常形式の PRE アルゴリズムにも利用できる。実際に本研究では、比較実験用に Knoop らのアルゴリズムである Lazy Code Motion[12] をこの手法で SSA 形式に対応させて実装した。

2 実行時情報を利用した PRE

本研究で提案する方法は [3] に触発されたものであるが、以下のような特徴がある。

- 実行時情報の利用
- コピー伝播との組み合わせ
- SSA 形式への対応

以下、2、3、4 章で順を追って説明する。

本手法の特徴の一つ目として、実行時情報を利用するということがあげられる。本章では、実行時情報を利用した部分冗長除去のアルゴリズムを説明する。基本的な手順は以下のとおりである。

1. Divided block graph の作成

* 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

*1 以後 PRE と呼ぶ

*2 以後 SSA 形式と呼ぶ

2. データフロー解析
3. Reduced graph の作成
4. Single-source, single-sink graph の作成
5. 最小カットの計算
6. コードの移動

以下、若干の議論ののち、この手順に沿って説明する。

2.1 保守的な方法と積極的な方法

通常の PRE アルゴリズムは**保守的な方法 (conservative method)** である。保守的な方法とは、計算回数を増加させる可能性のあるプログラムポイントには式を挿入しない方法である。一方、計算回数が増加する可能性があっても式の挿入を行う方法を**積極的な方法 (aggressive method)** と呼ぶ。

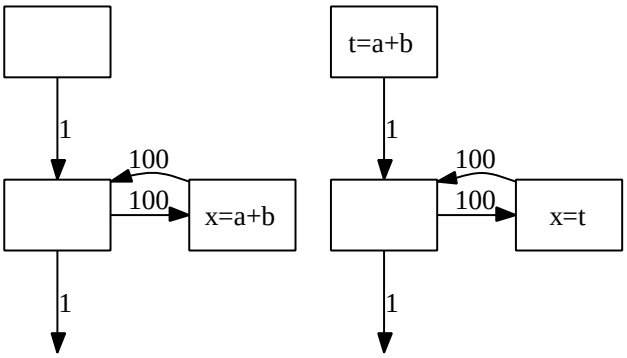

図 1　積極的な方法が有効な例

図 1 の制御フローグラフは辺に実行時に通った回数を表示してある。以後同様に、制御フローグラフの辺に表示する数字は実行時に通った回数である。図 1 は積極的な方法が有効な例である。このような変換は保守的な方法では、もし 1 度もループしなかった場合計算回数が増加するため行わないが、実際には変換前は式 $a + b$ の計算回数は 100 回だったのに対して、変換後は 1 回となっている。

一方、図 2 は積極的な方法が不利益な例である。このような変換も、保守的な方法では計算回数が増加する可能性があるため行わない。変換前は式「$a + b$」の計算回数は 20 回だったのに対して、変換後は 81 回となっている。

2.2 実行時情報を利用する方法

実行時情報を利用して部分冗長除去をおこなうと、より実行時の計算量を小さくするような PRE が可能となる。実行時情報を利用した解析を行うことによって、図 1 のようなプログラムでは PRE を行い、図 2 のようなプログラムでは PRE を行わないようにする。このように、実行時情報を利用した PRE は、式の計算回数を最小にするようなコード移動を行う。

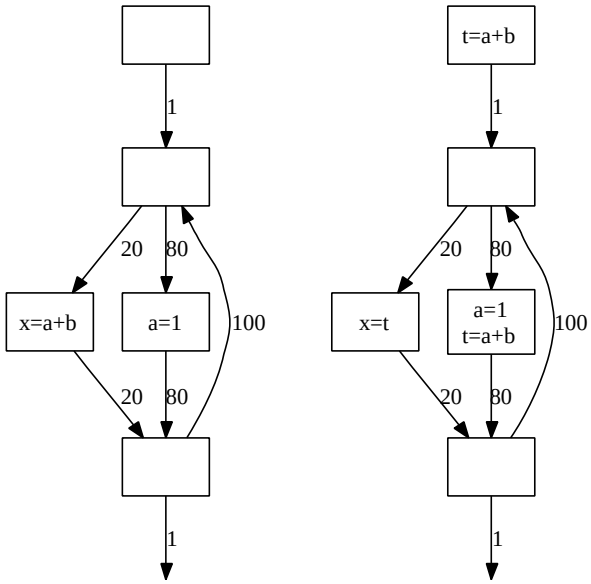

図 2　積極的な方法が不利益な例

2.3 実行時情報の収集

本研究では実行時情報を集めるために、対象のプログラムに対して、実行時情報を集めるための命令を埋め込んで予め一回実行しておく。埋め込む命令は C 言語[*3]で記述し、コンパイル時に対象のプログラムを変換したものと一緒に読み込む。実行時に辺を通った回数を記録したファイルがプロファイルとして出力される。

2.4 Divided block graph の作成

ここからは PRE の対象となっている各式一つ一つに対して行なう処理の話となる。以下では例として式「$a + b$」を用いる。本手法では各式に対して以下説明するようなグラフの作成を行なう。

本手法の元になっている [3] で提案されている実行時情報を利用したアルゴリズムでは、データフロー解析を行なうグラフのノードは文となっている。しかし、文をノードとした解析は解析時間が長くかかるので、本手法では基本ブロックを必要最小限だけ分割したものをノードとしたグラフを作成し、データフロー解析の対象とする。このグラフを **divided block graph** と呼ぶ。なお、基本ブロックは最大でも 2 個にしか分割しない。

本アルゴリズムは後で説明するように、グラフのカットによって切断された辺に計算式を挿入するものである。そこで、命令を挿入する可能性のある場所で基本ブロックを分割しておく。分割の場所は、基本ブロックに変更文[*4]が存在した場合、その最後の変更文の直後である。図 3 に

[*3] 実験対象とするソースプログラムは C 言語である
[*4] 対象とする式のオペランドを変更する文のこと

divided block graph の作成の例を示す。

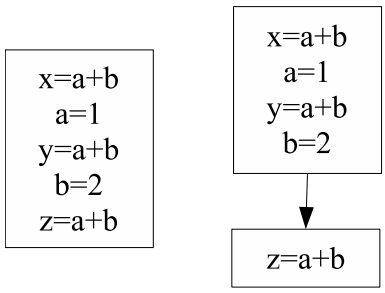

図 3　Divided block graph

図 3 (左) の変更文は「$b = 2$」である。この変更文の直後で分割を行なったものが図 3 (右) であり、これが divided block graph である。

2.5　データフロー解析

本手法のデータフロー解析に使用する局所的な性質を以下に示す。

定義 2.1 (Transparency) ある式に対して、ブロック i の命令がその式のオペランドの値を変化させない場合、ブロック i でその式は **transparent** であるという。ブロック i の transparency を $Transp(i)$ と表す。

定義 2.2 (Local availability) ある式に対して、

- ブロック i に少なくとも一つその式が存在する
- その式の最後の計算の後に現れる命令が、その式のオペランドの値を変化させない

これらを満たすとき、ブロック i でその式は **locally available** であるという。ブロック i の local availability を $Comp(i)$ と表す。

定義 2.3 (Local anticipability) ある式に対して、

- ブロック i に少なくとも一つその式が存在する
- その式の最初の計算の前に現れる命令が、その式のオペランドの値を変化させない

を満たすとき、ブロック i でその式は **locally anticipable** であるという。ブロック i の local anticipability を $Antloc(i)$ と表す。

次に、本手法のデータフロー解析に使用する大域的な性質を以下に示す。以下でプレフィックス N はブロックの入口、X は出口を表す。

定義 2.4 (Availability)

$$NAval(i) = \begin{cases} false & i \text{ is entry block} \\ \bigwedge_{j \in Pred(i)} XAval(j) & \text{otherwise} \end{cases}$$

$$XAval(i) = Comp(i) \vee (NAval(i) \wedge Transp(i))$$

定義 2.5 (Partial anticipability)

$$XPant(i) = \begin{cases} false & i \text{ is exit block} \\ \bigvee_{j \in Succ(i)} NPant(j) & \text{otherwise} \end{cases}$$

$$NPant(i) = Antloc(i) \vee (XPant(i) \wedge Transp(i))$$

さらにこれらを使用して、次に説明する reduced graph の作成に必要な辺に関する性質を計算する。計算する性質を次に示す。

- *InsRedund*: この辺に式を挿入するとその式が冗長となる。
- *InsUseless*: この辺に式を挿入しても決して後で使用されない。
- *NonEss*: この辺に式の挿入を行なっても無益である。

辺 (u, v) に関するこれらの性質の計算式を次に示す。

定義 2.6 (辺に関する性質)

$$InsRedund(u, v) = XAval(u)$$
$$InsUseless(u, v) = \neg NPant(v)$$
$$NonEss(u, v) = InsRedund(u, v) \vee InsUseless(u, v)$$

2.6　Reduced graph の作成

式の挿入を行う可能性のない辺は部分冗長除去における解析には不要である。そこで、不要な辺（NonEss が真である辺）および不要なブロック（接続する辺が不要な辺になったブロック）を除去したグラフを作成する。このグラフを **reduced graph** と呼ぶ。図 4（左）のグラフから不要な辺を除去すると図 4（右）の辺だけが残り、それに伴い下の 2 つのブロックも除去される。

2.7　Single-source, single-sink graph の作成

Reduced graph に新しい入口ブロックと出口ブロックを追加し、**single-source, single-sink graph** を作成する。さらに、先行ブロックのないブロックに新しい入口ブロックから辺をのばし、後続ブロックのないブロックから新しい出口ブロックに辺をのばす。図 5 (右) が図 5 (左) の reduced graph から作成された single-source, single-sink graph である。このとき、新しく引かれた辺の重みは無限大にする。

2.8　最小カットの計算と部分冗長性の除去

Single-source, single-sink graph は重みを容量として考えると、フローネットワークとして見ることができる。こ

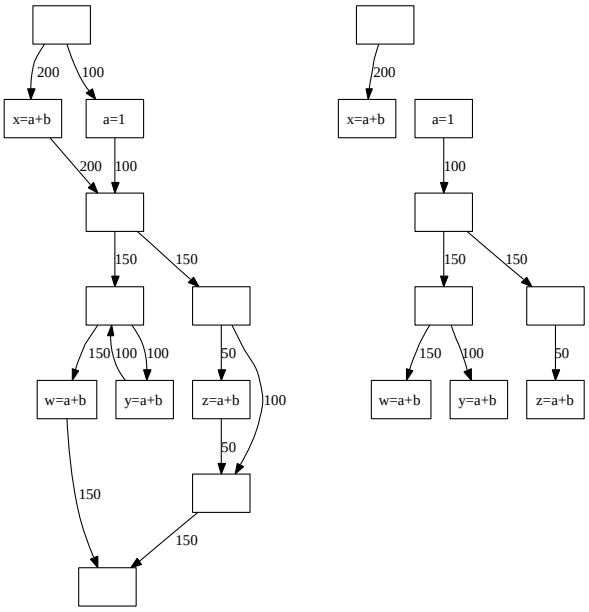

図 4　Reduced graph

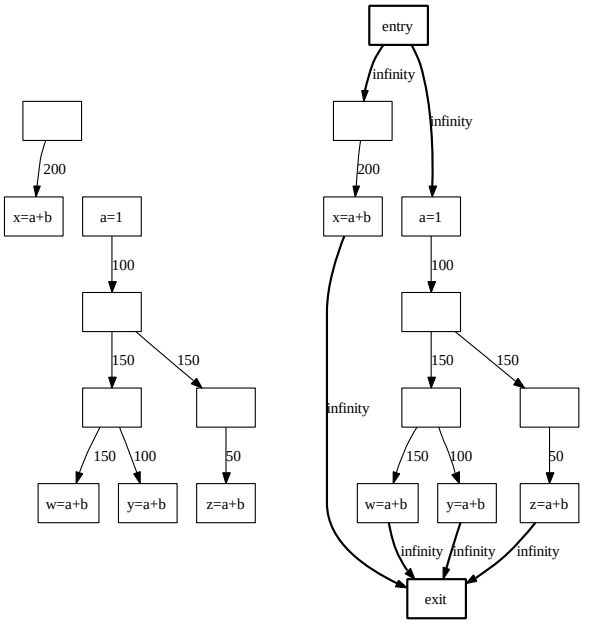

図 5　Single-source, single-sink graph

のフローネットワークに対してカットを作成し、作成したカットによって切断される辺に式「$a+b$」を挿入すると、元からあった式「$a+b$」は全て冗長となる。また、挿入した式が実行時に計算される回数はカットの容量となる。したがって、single-source, single-sink graph に対して最小カットを計算し、最小カットによって切断された辺に式「$a+b$」を挿入し、元からあるその式「$a+b$」の計算を除去することによって、実行時の式「$a+b$」の計算回数を最小にするような部分冗長除去を行うことができる。図 5（右）の single-source, single-sink graph の最小カットは図 6 のようになる。図 6 は、カットによって切断される辺を点線で示してある。元のプログラム（図 4）のこのカットによって切断された辺に式「$a+b$」の計算を挿入し、冗長な式を置き換えたものが図 7 であり、このとき実行時に式「$a+b$」が計算される回数は 300 回である。

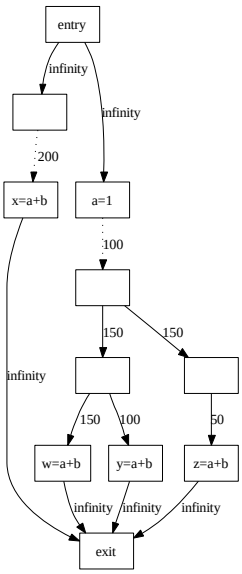

図 6　最小カット

3　コピー伝播と組み合わせた PRE

本研究では、特徴の 2 つ目にあげたようにコピー伝播と PRE を組み合わせた最適化を行う。図 8 (a) に現れるような長い式は、(b) のように短い式に分解されるのが一般的である。このような式を移動したとき、実際には式がなくなるのではなく、(c) のように冗長となった式を一時変数で置き換えることになる。これによって生じたコピー文「$t = p1$」は、残りの式「$t+c$」の移動を妨げることになる。このような式「$t = p1$」をすべて動かすのは難しい。そこで本研究では (d) のように、PRE の結果生じたコピー文「$t = p1$」に対してコピー伝播を行うことによってすべて

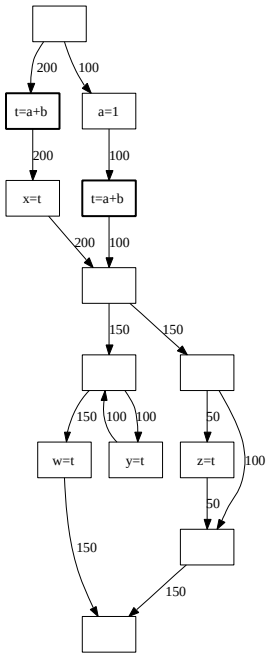

図 7　実行時情報を利用した PRE

の式を移動させることができる。その結果が (e) である。

ただし、本手法は SSA 形式に対応させるために、CSSA 形式上で行なう必要がある[*5]。通常のコピー伝播を行なうと CSSA 形式が壊れてしまう可能性があるので、ここでは、ブロック内に限定してコピー伝播を行なうこととする。したがって、コピー文は不要命令にはならないのでここでは除去することができないが、PBPRE の後に通常のコピー伝播と不要命令除去を行なえば、これらのコピー文は除去することができる。

ここで提案する「コピー伝播と組み合わせた RRE」と PRE の前後にコピー伝播を行なうこととの違いは、「連鎖的である」ということである。下のアルゴリズムに示したように、本手法は PRE とコピー伝播を繰り返すことによって、冗長性を除去したことによって新たに発生した冗長性をも除去することができる。

以下にコピー伝播と組み合わせた PRE のアルゴリズムを示す。

[*5] 4 節参照

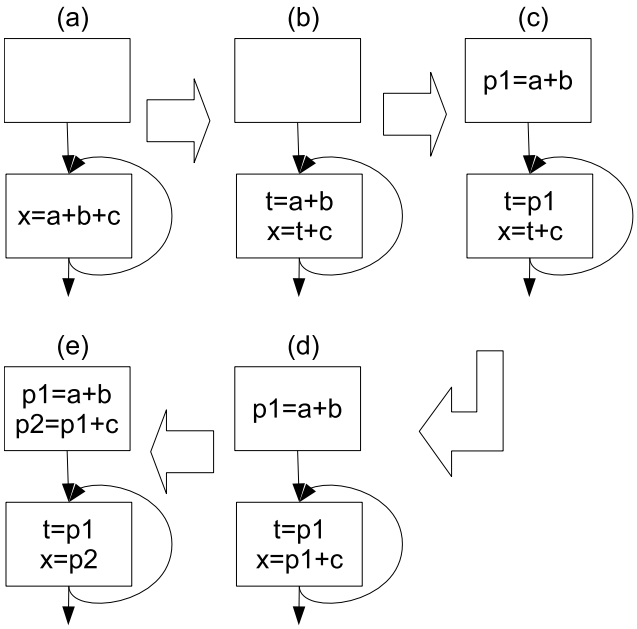

図 8　コピー伝播を用いた PRE

アルゴリズム 3.1 (コピー伝播と組み合わせた PRE)

$cpypPre(Flowgraph\ fg)\{$
　$exprList := fg$ 中に現れる式のリスト;
　for(各式 $expr \in exprList$)$\{$
　　$expr$ に対して PRE を行なう;
　　PRE によって生じたコピー文に対して
　　コピー伝播を行なう;
　　if(コピー伝播によって生じた式が
　　$exprList$ に存在しない)$\{$
　　　コピー伝播によって生じた式を
　　　$exprList$ に加える;
　　$\}$
　$\}$
$\}$

4　SSA 形式上の PRE

本手法の元になっている [3] は通常形式のアルゴリズムである。特徴の 3 つ目にあげたように、本研究ではそれを SSA 形式に適用した。また、[12] のアルゴリズムの SSA 化も行なったが、その方法は一般性のある手法になっている。

4.1　背景

PRE を SSA 形式のプログラムに適用するのは簡単ではない。たとえば、図 9（左）の「$a + b$」は、PRE の対象となるものであるが、図 9（右）のように SSA 形式にすると、2 つの「$a + b$」が別々の形になってしまうため同一の式で冗長であることが認識できなくなってしまう。

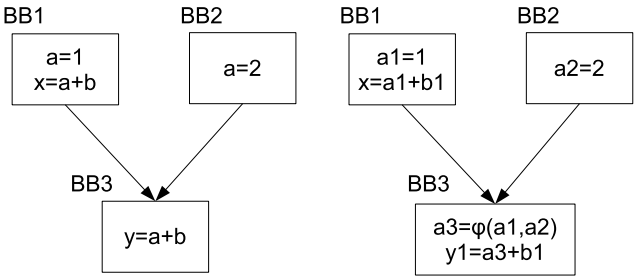

図 9　PRE と SSA 形式

また、仮に「$a1+b1$」と「$a3+b1$」が同じだと認識できても、図 9（右）の BB3 の「$a3+b1$」をそのままの形で BB2 に移動することはできない。$a3$ の使用を定義の先行ブロックに移動することはできないからである。

4.2　CSSA 形式と TSSA 形式

CSSA 形式とは、Cytron らのアルゴリズム [4] で SSA 形式に変換した直後の SSA 形式のことをいう。CSSA 形式の特徴として、同じ **phi congruence class (PCC)** [*6]に属する全ての変数を同じ代表変数に置き換え、全てのφ関数を除去してしまっても元のプログラムの意味を変えないということが保証されている [18]。しかし、SSA 形式での最適化変換を行うと、このような CSSA 形式の性質は一般には保たれない。つまり、φ関数で結ばれた変数の間に生存区間の干渉が発生する。このような SSA 形式を**TSSA 形式**という。

4.3　PCC による式の識別

PCC とは、φ関数で結ばれた変数の集合である。たとえば図 9（右）では、$\{a1, a2, a3\}$ が PCC である。Phi congruence class を作成する方法を次に示す。

アルゴリズム 4.1 (PCC の作成)

```
makePhiCongruenceClass(FlowGraph fg){
    for(各φ関数 phi ∈ fg){
        phi で結ばれた変数を同じ PCC にする;
    }
    共通部分を持つ PCC をマージする;
}
```

また、PCC を使用して式の同一性を識別する方法を次に示す。

[*6] 以後 PCC と呼ぶ

アルゴリズム 4.2 (PCC による式の識別)

```
same(expr1, expr2){
    if(式 expr1, expr2 の演算子、
        型がそれぞれ等しい){
        if(expr1, expr2 のオペランドが属する
            PCC がそれぞれ等しい){
            return true;
        }
    }
    return false;
}
```

CSSA 形式において $same(expr1, expr2) = true$ ならば、$expr1$ と $expr2$ は同一の式であると認識する。

本手法では、このような CSSA 形式上での PCC の性質を利用して、式のそれぞれのオペランドが同じ PCC に属する場合、それらの式を同一の式と認識することにする。そのために、本システムでは PRE を行う前には、TSSA 形式から CSSA 形式への変換 [18] を行っている。

通常形式の PRE アルゴリズムでは、字句等価な式をリストにしてリスト内のそれぞれの式に対して PRE を適用するが、本手法では、PCC の情報を用いて同一と認識された式の内から任意の 1 つを選びリストの要素とする。これによって、通常形式上と同様に冗長性を認識することができる。この *same* メソッドを使用して、制御フローグラフから PRE の対象となる式をリストにする方法を次に示す。

アルゴリズム 4.3 (対象となる式のリストの作成)

```
makeExprList(FlowGraph fg){
    exprList を初期化
    for(各式 expr1 ∈ fg){
        boolean found := false;
        for(各式 expr2 ∈ exprList){
            if(same(expr1, expr2)){
                found := true;
                break;
            }
        }
        if(!found){
            exprLisr.add(expr1);
        }
    }
    return exprList;
}
```

図 10 は SSA 形式上で PRE を行った例である。以下、この例を用いて各ステップを説明する。

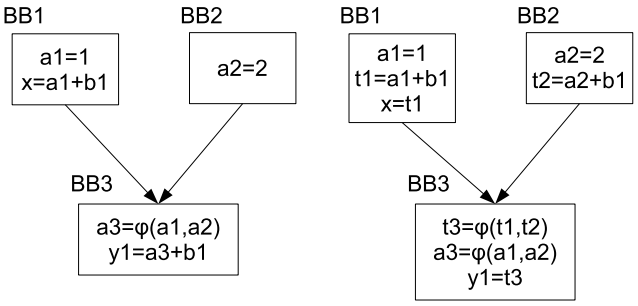

図 10　SSA 形式上の PRE

4.4　式の挿入

式の挿入点が決定したら、挿入する式の変数を決めなければならない。式は PCC の情報を用いて同一とみなされた式のうちの 1 つなので、このまま挿入することはできない。

そこで、式のそれぞれの変数に対して、挿入点からその点を支配する点を上向きに辿っていき、その変数と同じ PCC に属する変数の定義を見つけたら、その見つかった変数を挿入する式の変数とする。式の値を保存するための一時変数は、まだ定義に使用されていない任意の変数を指定すればよい。

例えば、図 10（左）の BB1 に「$a3+b1$」に相当する式を挿入したいときは、そこから上を探すと $a3$ の PCC に属する $a1$ の定義「$a1=1$」が見つかるので、それを用いて「$a1+b1$」を挿入する．また、4.6 節の名前替えで使用するために、挿入した式は記憶しておく。

以下に式 $expr$ をプログラムポイント p に挿入するアルゴリズムを示す。$expr$ は $a\ op\ b$ とする。op は演算子、a, b はオペランドである。

アルゴリズム 4.4 (式の挿入)

$insertExpr(expr, p)$\{
　p を支配する点を上向きに辿り、
　a と同じ PCC に属する変数 a'の定義を見つける;
　p を支配する点を上向きに辿り、
　b と同じ PCC に属する変数 b'の定義を見つける;
　まだ定義されてない変数 t_iを作成する;
　代入文 $t_i = a'\ op\ b'$を p に挿入する;
\}

4.5　φ関数の挿入

式の値を保存した一時変数は SSA 形式に従って定義されている。これらの一時変数の定義が合流するプログラムポイントにφ関数を挿入する必要がある。本手法では、**最小 SSA** を作成するアルゴリズム [4] に従って、式を挿入したブロックの支配辺境に一時変数のためのφ関数を挿入する。φ関数の左辺は、実際には SSA 形式に従った任意の名前をつければよい。φ関数の引数の変数にはこの時点では任意の名前をつけておき、4.6 節で名前替えする。また、この名前替えで使用するために、挿入したφ関数は記憶しておく。

以下にφ関数を挿入するアルゴリズムを示す。

アルゴリズム 4.5 (φ関数の挿入)

$insertPhi()$\{
　for(式の挿入を行なった各ブロック blk)\{
　　if(blk の支配辺境 df にまだφ関数が
　　挿入されていない)\{
　　まだ定義されていない変数 t_iを作成する;
　　ダミーの変数として任意の変数 t_j, t_kを
　　作成する;
　　φ関数 $t_i = \phi\ (t_j, t_k)$ を df に挿入する;
　　\}
　\}
\}

たとえば図 10（右）では、式の値を保存した一時変数 $t1$ の支配辺境に「$t3 = \phi\ (t?, t?)$」を挿入する．

4.6　φ関数の引数の変数の名前替え

φ関数を正確に表すため「$t_i = \phi\ (t_j : L_a, t_k : L_b)$」と記す。これは、ブロック L_a からきたときは t_j を使用し、ブロック L_b からきたときは t_k を使用するものとする。引数の変数のコロンの後の L_a, L_b はブロックのラベルであり、そのラベルを持つブロックから訪れた場合、対応する変数が左辺に代入される。なお、この時点では、t_j, t_k には任意の名前がついている。t_j の名前を決めるには、ブロック L_a からブロック L_a を支配するブロックを上向きに辿っていき、4.4 節で挿入した式か、4.5 節で挿入したφ関数を見つけたら、その左辺を t_j の名前とする。t_k も同様である。また、新しいφ関数を作成したので、PCC も更新しておく。

以下に名前替えのアルゴリズムを示す。

アルゴリズム 4.6 (φ関数の引数の変数の名前替え)

$rename()$\{
　for(挿入した各φ関数 $t_i = \phi\ (t_j : L_a, t_k : L_b)$)\{
　　ブロック L_aからそのブロックを支配する
　　ブロックを上向きに辿り、
　　挿入した式かφ関数を見つけたら、
　　その左辺で t_jを置き換える;
　　ブロック L_bからそのブロックを支配する
　　ブロックを上向きに辿り、
　　挿入した式かφ関数を見つけたら、
　　その左辺で t_kを置き換える;
　　名前替えを行なったφ関数の引数の変数を
　　同じ PCC にする;
　\}
\}

4.7 冗長となった式の除去

冗長となった式の除去は、実際には式の一時変数への置き換えである。ここで、置き換える一時変数を決める必要がある。

対象とする式一つに対して、挿入された一時変数は全て同じ PCC に属している。また、この PCC は式の挿入を行なうフェーズからこのフェーズまで記憶しておく。よって、置き換える式の存在する点から、その点を支配する点を辿っていき、挿入された一時変数が属する PCC に属する変数の定義を見つけたら、その変数で式を置き換える。

たとえば図 10 では、挿入された一時変数の PCC は $\{t1, t2, t3\}$ である。BB3 の「$a3 + b1$」を一時変数に置き換えたいときは、支配する点を辿っていくと、一時変数と同じ PCC に属する変数 $t3$ の定義「$t3 = \phi\ (t1, t2)$」が見つかるので、$t3$ で置き換える。BB1 の「$a1 + b1$」を置き換えたいときは、$t1$ の定義「$t1 = a1 + b1$」が見つかるので $t1$ で置き換える。

以下にプログラムポイント p の右辺の式を一時変数で置き換えるアルゴリズムを示す。点 p はすでに最小カットの場所として求まっており、一時変数は t と同じ PCC に属しているとする。

アルゴリズム 4.7 (冗長となった式の除去)

$replace(p, t)$\{
　p を支配する点を上向きに辿り、
　t と同じ PCC に属する変数 t'の定義を見つける;
　p に存在する文の右辺の式を t'で置き換える;
\}

4.8 他の PRE アルゴリズムの SSA 形式への対応

本手法を要約すると、以下のような手順で通常形式の PRE アルゴリズムを SSA 形式に適用した。

1. TSSA 形式から CSSA 形式への変換
2. PCC の作成
3. 式の挿入
4. φ関数の挿入
5. φ関数の引数の変数の名前替え
6. 冗長となった式の除去

この手順のうち、3 から 6 は作成した PCC に基づいて行なう。

この SSA 形式への適用手法は一般的な枠組になっている。実際、本研究では、後で述べる比較実験のために、代表的な通常形式上の PRE アルゴリズムである Lazy Code Motion[12] を上で述べた手順に従って SSA 形式に対応させたものを実装した。同様に、本章の方法を使用すれば、SSA 形式に対応していない多くの PRE のアルゴリズムを SSA 形式に対応させることができる。このことは新しい知見である。

5 Profile-Based Partial Redundancy Elimination on SSA form

PBPRE 全体の流れについて以下に示す。

PBPRE 全体を通してのアルゴリズムを以下に示す。このアルゴリズムは入力の制御フローグラフ fg に対して変換を行なう。

アルゴリズム 5.1 (PBPRE)

pbpre(*Flowgraph fg*){
　実行時情報の読み込み;
　危険辺の除去;
　TSSA 形式から *CSSA* 形式への変換;
　phi congruence class の作成;
　PRE の対象となる式のリスト *exprList* の作成;
　while(*exprList* が空でない){
　　式 *expr* := *exprList* の最初の要素;
　　exprList から *expr* を除去;
　　divided block graph の作成;
　　データフロー解析;
　　reduced graph の作成;
　　single-source, *single-sink graph* の作成;
　　最小カットの計算;
　　式の挿入;
　　φ関数の挿入;
　　φ関数の引数の変数の名前替え;
　　冗長な計算の置き換え;
　　置き換えによって生じたコピー文にコピー伝播;
　　コピー伝播によって生じた式が *exprList* に
　　存在しなければ *exprList* に追加;
　}
}

ここで分かるように、このアルゴリズムは対象となる式に対して 1 つずつ適用する。このアルゴリズムについてのより詳しい説明は [9] で行なわれている。

6 評価

実験には COINS の SSA 形式最適化コンパイラ用モジュール [16] を用い、Sun Blade 1000 (UltraSPARC III) で実行時間を測定した。評価した最適化はすべて SSA 形式上のものである。

効果を確認するためのベンチマークプログラムとして、6 個の小さなプログラムと、SPEC CINT2000 の 3 個のベンチマークを使用して、各最適化を行なった。

使用した小さなベンチマークプログラムは以下の 6 個である。

- バブルソート (bubble sort)
- 挿入ソート (insertion sort)
- 安定な結婚の問題 (marriage)
- n 女王問題 (nqueens)
- クイックソート (quick sort)
- 選択ソート (selection sort)

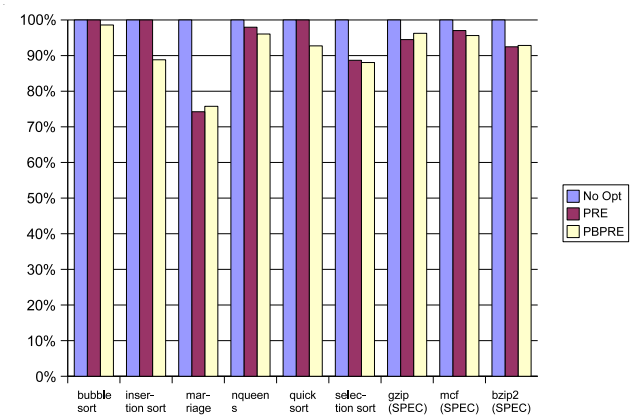

図 11　PBPRE 単体の実行時間

また、SPEC からは以下の 3 個を使用した。

- 164.gzip (gzip)
- 181.mcf (mcf)
- 256.bzip2 (bzip2)

SPEC にはベンチマークを行なう際の入力データとして、test、train、ref がある。本実験では、小さな test データで実行して実行時情報を集め、大きな ref データで実行時間の測定を行なった。

6.1 PBPRE を単体で行ったときの評価

まず、PBPRE 単体の評価を行う。しかし PBPRE は、PBPRE を行った後のフェーズでコピー伝播と不要命令除去を行うことを前提にしているので、これらの最適化は常に行う。このことを踏まえ、比較対照は次のようにする。

- **No Opt**：SSA 変換 → SSA 逆変換
- **PRE**：SSA 変換 → PRE → コピー伝播 → 不要命令除去 → SSA 逆変換
- **PBPRE**：SSA 変換 → PBPRE → コピー伝播 → 不要命令除去 → SSA 逆変換

この比較で用いる PRE は、Knoop らの方法 [12] を、4 節の方法によって SSA 形式に対応させたものである。

図 11 から分かるように、全てのプログラムで PBPRE による最適化の効果が現れており、最大で 25% ほど実行時間が短縮されている。PRE と比べると、2 個のプログラムで実行時間に 10% ほどの改善が見られた。

また、コンパイル時間だが、本実験は Sun Blade で行なったが、小さなプログラムでは数秒、SPEC の大きなプログラムでは数時間となっている。Pentium4 2.8GHz のマシンでは SPEC のコンパイルは数十分であった。コンパイル時間を重要視した実装ではないため、実装の技術によってコンパイル時間は短縮できる可能性がある。

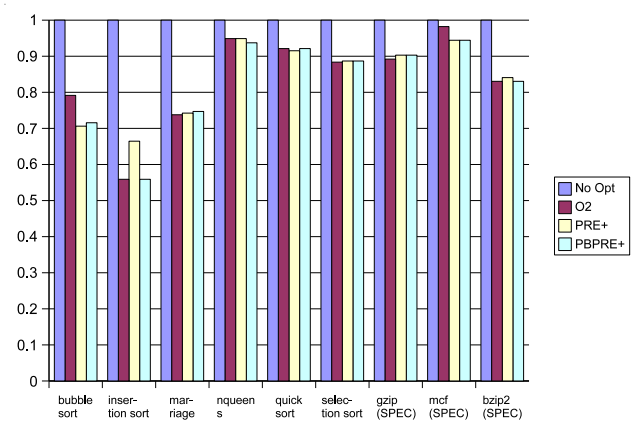

図 12　PBPRE をさまざまな最適化と共に行ったときの実行時間

6.2　PBPRE をさまざまな最適化と共に行ったときの評価

次に、PBPRE をさまざまな最適化と共に行ったときの評価を行う。比較対照は次のようにする。

- **No Opt**：SSA 変換 → SSA 逆変換
- **O2**：SSA 変換 → 危険辺の除去 → 共通部分式除去 → 定数伝播 → ループ不変式移動 → 演算の強さの軽減 → ループ不変式移動 → 定数伝播 → コピー伝播 → PREQP → 定数伝播 → 無用 φ 命令除去 → 不要命令除去 → SSA 逆変換
- **PRE+**：SSA 変換 → 危険辺の除去 → 共通部分式除去 → 定数伝播 → ループ不変式移動 → 演算の強さの軽減 → ループ不変式移動 → 定数伝播 → コピー伝播 → PRE → 定数伝播 → 無用 φ 命令除去 → 不要命令除去 → SSA 逆変換
- **PBPRE+**：SSA 変換 → 危険辺の除去 → 共通部分式除去 → 定数伝播 → ループ不変式移動 → 演算の強さの軽減 → ループ不変式移動 → 定数伝播 → コピー伝播 → PBPRE → コピー伝播 → 定数伝播 → 無用 φ 命令除去 → 不要命令除去 → SSA 逆変換

この評価では、現在 COINS の SSA 最適化で最もよい結果の出ている最適化の組み合わせであり、今後 O2 オプションとして採用される予定のものを比較対象とし、便宜的に「O2」と表記する。この O2 で行われる PREQP とは、滝本の**質問伝播に基づく大域値番号付けと部分冗長除去**である [16]。この PREQP を PRE, PBPRE と置き換えたものをそれぞれ PRE+, PBPRE+ とし、実行時間によって評価する。

図 12 から分かるように、PBPRE は O2 と比べて、ほぼ同等、または効率が良いコードを出す。bubble sort では、O2 に比べ PBPRE が実行時間において 10% ほど短い。ただ、PRE と PBPRE には差がないことから、これは O2 にたまたまなんらかのオーバーヘッドがかかったものかもしれない。mcf では PBPRE は O2 より 4% ほど効率が良いコードを出す。PBPRE は PRE と比べて、ほぼ同等、または効率の良いコードを出す。insertion sort では、PRE に比べ PBPRE が実行時間において 10% ほど短い。

7　関連研究との比較

7.1　従来の SSA 形式上の PRE

[10, 19] では SSA 形式上の PRE が提案されている。しかし、これらのアルゴリズムは制御フローグラフとは別の特別なグラフの作成と、そのグラフにおける複雑な解析が必要である。また、[2] のアルゴリズムは処理中に SSA 形式から通常形式に戻ってしまう。

本手法で提案する SSA 形式への拡張法は、特別なグラフの作成を必要としない。そのため、解析時間で得をしている。また、処理中に通常形式に戻ってしまうこともない。

ところで、PRE で行なうデータフロー解析を高速に行なう手法として、**ビットベクタ (bit-vector)** 法というものがある [1]。ビットベクタ法は、データフロー情報の集合をビットベクタで表現することによって、1 ワード分のデータフロー計算を並列に行なう方法である。ビットベクタ法を使用した通常形式の PRE アルゴリズムを本手法で提案する方法で SSA 形式に拡張すれば、解析時間において従来の方法を上回るものができる可能性がある。

残念ながら、本手法である PBPRE は、元となっている通常形式のアルゴリズムが式ごとに最小カットを求める方法であるため、ビットベクタ法を使用することができない。そこで、PBPRE の解析をビットベクタ法で行なうことはできない。

7.2　実行時情報を用いる PRE

実行時情報を用いる PRE には [3, 8, 17] などがある。

本アルゴリズムの多くの部分は [3] のアルゴリズムによるものである。[3] との違いは、[3] は文を対象にデータフロー解析を行なっていたのに対して、本アルゴリズムはブロックを対象にデータフロー解析を行なうということである。これによって、デーフロー解析を高速に行なうことができる。

また、[3] では実行時情報として、辺を通った回数を使用しているのに対して、[17] はブロックを通った回数を使用する。ブロックを通った回数を使用する方法だと、実行時情報を比較的容易に集めることができるという利点がある。

[8] は複雑な形で実行時情報を集めるので、上の 2 つに比べて実行時情報を集めるのにコストがかかる。

8 おわりに

本研究では、PBPRE という新しい PRE アルゴリズムを提案した。このアルゴリズムは次のような特徴を持ったアルゴリズムである。

- 実行時情報を利用することによって、保守的な方法と積極的な方法の長所を合わせたコード移動を行うことができる。
- コピー伝播を組み合わせることによって、今まで移動が困難であった式を移動させることができる。
- SSA 形式のまま処理を行うことができる。

これによって、一部のプログラムでは既存の PRE に比べて実行時間に改善が見られた。また、COINS で現在最も良い結果を出している最適化の一部を PBPRE に置き換えて実験を行ったが、ここでも一部のプログラムで効果が見られた。よって、既存の PRE や O2 と同等、もしくはものによってすぐれており、採用する価値がある。

また、4 章で述べた方法を用いると、通常形式上の多くの PRE のアルゴリズムを SSA 形式に対応させることができる。これは新しい知見であり、SSA 形式のまま最適化できること、およびビットベクタ法の採用により解析時間の短縮が見込まれるなど、有用性が高い。

参考文献

[1] Appel, A. W. and Palsberg, J. “Modern Compiler Implementation in Java 2nd ed.”. Cambridge University Press, New York, NY, USA, 2002.

[2] Briggs, P. and Cooper, K. D. “Effective partial redundancy elimination”. In *PLDI ’94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pp. 159–170, New York, NY, USA, 1994. ACM Press.

[3] Cai, Q. and Xue, J. “Optimal and efficient speculation-based partial redundancy elimination”. In *CGO ’03: Proceedings of the international symposium on Code generation and optimization*, pp. 91–102, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N. and Zadeck, F. K. “Efficiently computing static single assignment form and the control dependence graph”. *ACM Trans. Program. Lang. Syst.*, Vol. 13, No. 4, pp. 451–490, 1991.

[5] Dhamdhere, D. M. “A fast algorithm for code movement optimisation”. *SIGPLAN Not.*, Vol. 23, No. 10, pp. 172–180, 1988.

[6] Dhamdhere, D. M. “Practical adaption of the global optimization algorithm of Morel and Renvoise”. *ACM Trans. Program. Lang. Syst.*, Vol. 13, No. 2, pp. 291–294, 1991.

[7] Dhamdhere, D. M. “E-path_PRE: partial redundancy elimination made easy”. *SIGPLAN Not.*, Vol. 37, No. 8, pp. 53–65, 2002.

[8] Gupta, R., Berson, D. A. and Fang, J. Z. “Path Profile Guided Partial Redundancy Elimination Using Speculation”. In *ICCL ’98: Proceedings of the 1998 International Conference on Computer Languages*, p. 230, Washington, DC, USA, 1998. IEEE Computer Society.

[9] 伊藤陽. “実行時情報を利用した部分冗長除去”. Master’s thesis, 東京工業大学 大学院 数理・計算科学専攻, 2006.

[10] Kennedy, R., Chan, S., Liu, S.-M., Lo, R., Tu, P. and Chow, F. “Partial redundancy elimination in SSA form”. *ACM Trans. Program. Lang. Syst.*, Vol. 21, No. 3, pp. 627–676, 1999.

[11] Knoop, J., Rüthing, O. and Steffen, B. “Lazy code motion”. In *PLDI ’92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pp. 224–234, New York, NY, USA, 1992. ACM Press.

[12] Knoop, J., Rüthing, O. and Steffen, B. “Optimal code motion: theory and practice”. *ACM Trans. Program. Lang. Syst.*, Vol. 16, No. 4, pp. 1117–1155, 1994.

[13] 溝渕裕司, 立川英, 佐々政孝. “変更文の移動を可能にした静的単一代入形式上での部分冗長性除去”. 日本ソフトウェア科学会第 7 回プログラミングおよびプログラミング言語ワークショップ (PPL2005) 論文集, pp. 261–275, 2005.

[14] Morel, E. and Renvoise, C. “Global optimization by suppression of partial redundancies”. *Commun. ACM*, Vol. 22, No. 2, pp. 96–103, 1979.

[15] Paleri, V. K., Srikant, Y. N. and Shankar, P. “A simple algorithm for partial redundancy elimination”. *SIGPLAN Not.*, Vol. 33, No. 12, pp. 35–43, 1998.

[16] 佐々研究室. “静的単一代入形式最適化システム外部仕様書”, 2006. http://www.is.titech.ac.jp/~sassa/coins-www-ssa/japanese/ssa-external-japanese.pdf.

[17] Scholz, B., Horspool, N. and Knoop, J. “Optimizing for space and time usage with speculative partial redundancy elimination”. *LCTES '04, SIGPLAN Not.*, Vol. 39, No. 7, pp. 221–230, 2004.

[18] Sreedhar, V. C., Ju, R. D.-C., Gillies, D. M. and Santhanam, V. “Translating Out of Static Single Assignment Form”. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis, Lecture Notes in Computer Science*, Vol. 1694, pp. 194–210, London, UK, 1999. Springer-Verlag.

[19] 滝本宗宏, 原田賢一. “拡張値グラフに基づく効果的な部分冗長除去法”. 情報処理学会論文誌, Vol. 38, No. 11, pp. 2237–2250, 1997.