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

溝渕 裕司 [†]　　立川 英 [††]　　佐々 政孝 [†]
Yuji MIZOBUCHI　　Suguru TACHIKAWA　　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
[††] 株式会社 NTT データ
NTT DATA CORPORATION
{mizobuc0@is.titech.ac.jp, tachikawas@nttdata.co.jp, sassa@is.titech.ac.jp }

1 はじめに

本論文では、Knoop らによる通常形式上の怠けたコード移動を SSA 形式上に自然な形で適用し、また従来の SSA 形式上での部分冗長性除去では最適化効果のなかったプログラムに対して、変更文 (計算式で使用される変数の定義文) を移動することによって最適化効果を促進する方法を提案する。

1.1 背景

コード移動と呼ばれる最適化の技術は、実行時の不必要な再計算の回避によりプログラムの効率を改善する技術である。なかでも、*PRE* (*Partial Redundancy Elimination*、**部分冗長性除去**) は、共通部分式除去やループ不変コードのループ外移動を統合的に行う最適化手法で、非常に強力である。

近年では、*SSA* **形式**(*Static Single Assignment Form*、**静的単一代入形式**) での PRE (SSAPRE) を行う研究が盛んに行われてきている。

従来の部分冗長性除去で最適化効果を妨げる原因のひとつとして、部分冗長な式が用いる変数の定義文 (**変更文**) の存在が考えられる。部分冗長性除去では、プログラムのコード移動を行うが、上方へのコード移動の際に変更文を越えた移動はできない。そのため、変更文の許す範囲でのみコード移動が可能であり、最適化効果に限度がある。

立川の方法 [15] では、**怠けたコード移動**(*lazy code motion*) [8, 9] とよばれる、レジスタ圧力を軽減させる部分冗長性の除去のアルゴリズムを基にし、これを SSA 形式に適応させる試みを行っている。しかし、対象とできるプログラムは、同一 ϕ 関数の内で変数の生存区間の重なりのないもの、という制約があった。

また、その他の従来のアルゴリズム [5, 7, 14] では，制御フローグラフ上のみでは処理ができず、複雑な解析と特別なグラフの作成が必要であり、実装がかなり困難とされている。

いずれの方法も制約があったり複雑であり、変更文を移動しないことを前提とした最適化であって、変更文によってコード移動を妨げるプログラムに対して最適化効果を発揮できない。

1.2 概要

本研究では、変更文を移動する部分冗長性除去として、次のような条件を満たすようなアルゴリズムを提案する。

1. SSA 形式のまま処理できる自然なアルゴリズム
2. 制御フローグラフ上のみでの処理
3. 変更文の移動によるより最適な計算式のコード移動
4. 同一 ϕ 関数の中で変数の生存干渉のあるプログラムに対処出来る等の適用範囲の広いアルゴリズム

本研究では、Knoop らが提案した *lazy code motion* とよばれる部分冗長性除去のアルゴリズム [8] を SSA 形式に自然に拡張することにより、変更文の移動と同時に部分冗長性除去の効果を高める最適化アルゴリズムを開発した。これを、COINS コンパイラ・インフラストラクチャ [11] 上で実装し、アルゴリズムの有用性について検証を行った。

2 準備

2.1 SSA 形式について

2.1.1 概要

SSA 形式は，プログラム中間表現の表現形式の一つで、プログラム上の各変数の使用に対して、その使用に対する定義が字面上一ヶ所しかないように表現したものである。SSA 形式を中間表現として用いると、変数の定義と使用の関係が明確に表現され、最適化の実現容易性と最適化の実行効率が向上する。

2.1.2 SSA 形式の定義

SSA 形式は、Alpern、Rosen 、Wegman、Zadeck [1, 12] らによって考案されたプログラムの表現形式の一つで、各代入文の左辺の変数に一意的な添字を付けることにより、各添字付き変数の字面上の単一代入性が保証されている。SSA 形式は、次の有用な性質を持つ [6]。

1. 各変数の使用には、唯一の定義が到達する。
2. 制御フローグラフ上のノード N で、変数 v の異なった定義が合流するとき、ϕ **関数**を N の先頭に挿入することで、それらの到達する v の値を区別することができる。

ここでいう ϕ 関数とは次のような疑似的な関数である [6] 。$v_0 = \phi(v_1, v_2, \ldots)$ なる ϕ 関数 (図 1) を考える。ϕ

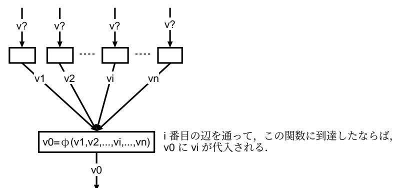

図 1: ϕ 関数

関数のパラメタ "v_1、v_2、..." はこの ϕ 関数が挿入されたノード N の先行ノードの数だけ存在する。この時、N の i 番目の辺に沿って v の定義が N に到達したならば、ϕ 関数は v_i を返す。

2.2　部分冗長性除去について

2.2.1　冗長性

冗長な計算を除去する手法は、コンパイラのコード最適化において強力な手法であり、以前から多くの研究がなされてきた。

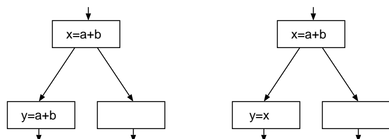

図 2: 冗長と冗長除去

図 2に示す 2 つの制御フローグラフは、説明を簡単にするために $a+b$ の計算に関係のある文 ($a+b$ の計算と、a や b の変更文) 以外の文は省略してあるものとする (以下同様) 。

図 2 (左) に示す制御フローグラフにおいて、下側の式の $a+b$ のように、プログラムの開始点から式に到達するすべての経路上に同じ計算式が存在するとき，その式は **冗長** (*redundant*)、あるいは**全冗長**であるという。このとき、図 2 (右) のように，冗長な式を除去することによってプログラムの意味を変えずに実行時の効率を上げることができる。これを **共通部分式の除去** (*common subexpression elimination*) という。

2.2.2　部分冗長性

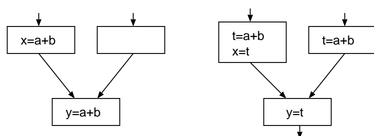

図 3: 部分冗長と冗長への変形

図 3 (左) の場合、左上のブロックからの経路上では、$a+b$ が 2 回計算される。しかし、下側の式を除去してしまうと、右上からの経路上に $a+b$ の式がなくなってしまうので、下側の $a+b$ は冗長ではない。この場合には、図 3 (右) のように、右上のブロックに $a+b$ を挿入することによって，下側の $a+b$ は全冗長になり、除去可能になる。このように一部の経路でのみ、冗長な式は **部分冗長** (*partially redundant*) であるという。

2.2.3　部分冗長性除去

部分冗長な式は、先行節に式を挿入することによって、冗長なものに変えることができる。新たな式の挿入点と冗長な式の決定は、部分冗長の考えを基にしたデータフロー解析によって実現することができる。

データフロー解析を用いて部分冗長を取り除く手法を**部分冗長性除去**(*partial redundancy elimination*，*PRE*) [10, 8, 9] という。図 3 (左) に対し部分冗長性除去を行うと図 3 (右) のようになる。

部分冗長性除去は共通部分式の除去に加えて **ループ不変コード移動**(*loop invariant code motion*) をも統一的に扱うことができる.

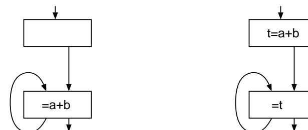

図 4: ループ不変コード移動

図 4 (左) で、ループの直前のブロックに $a+b$ を挿入することによって、ループ内部の $a+b$ は冗長となる。すなわち、ループ内部の $a+b$ は部分冗長であり、部分冗長性除去によってループ不変コード移動が実現できる。結果は図 4 (右) のようになる。

3　従来の SSAPRE と問題提起

SSA 形式上の部分冗長性除去を *SSAPRE* とよぶ。この節では従来の SSAPRE について説明し、その問題点を述べる。

3.1　従来の SSAPRE

部分冗長性除去を SSA 形式上のプログラムに適用するのは簡単ではない。

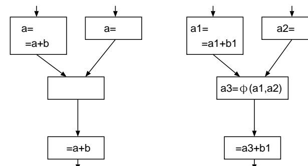

図 5: 部分冗長性除去と SSA 形式の例

例えば、図 5 (左) の $a+b$ は部分冗長性除去の対象となるものであり、一番下のブロックの $a+b$ を右上のブロックに移動することが望ましいが、図 5 (右) のような 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、同一の式でかつ、冗長であることが認識できなくなってしまう。また仮に a_1+b_1 と a_3+b_1 が同じだと認識できたとしても、図 5 (右) の一番下のブロックの a_3+b_1 をそのままの形で右上のブロックに移動することはできない。a_3 の使用をその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。

つまり具体的には、部分冗長性除去をこの SSA 形式上で処理しようとすると、

- 通常形式上で同一の変数として扱えたものが異なる変数として認識されてしまう。

- 異なる基本ブロックにコードを移動する際に変数がそのまま使用できない。

といった困難があるのである。

この問題をなんとか解決して、SSA 形式に対しても部分冗長性除去を行うアルゴリズムがいくつか考えられている [5, 7, 15]。例えば、立川の方法 [15] を 図5 (右) に対して適用すると、図 6 のような SSA 形式が得られる。図7には、別の例について同じ方法で SSAPRE をした例を示す。

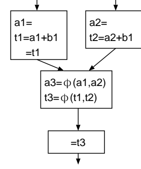

図 6: SSA 形式上の部分冗長性除去

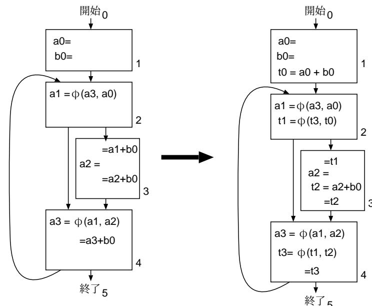

図 7: 従来の SSAPRE の最適化前と最適化後

3.2 問題提起

3.1節で説明をした従来の SSAPRE は、いずれも変更文 (すなわち部分冗長性除去の対象とする式のオペランドを変更する文) を移動させないことを前提とした手法であり、プログラムによっては、変更文が原因で最適化効果を得られないものもある。

例えば、図8は、図7のプログラム例の基本ブロック 4 の先頭に変更文を加えたものである。図7(右) では、計算式のコード移動がされており、計算回数も減らすことに成功している。しかし、図8(右) では、$a_1 + b_0$ の計算式に対してコード移動をしようとしても、直前にある a_1 の定義文に妨げられて移動させることができない。また、たとえ計算式の挿入ができたとしても、t_2 の使用は直後の t_2 のみで、$a_1 + b_0$ の計算回数を減らせていない。

これらの二つの例から、変更文が原因で最適化効果を小さくすることがあり、変更文を移動することで最適化効果の向上が期待できることがわかる。

この事実をふまえ、本研究では、変更文を移動させることにより、計算式のコード移動のできる範囲を広げ、より最適なプログラムポイントに式を移動することを可能にした新しい部分冗長性除去法を提案する。図 8の左の例に提案手法を適用すると図9のようになる。

4 本手法 (New PRE) のアルゴリズム

本手法は SSA 形式上で変更文を移動させることにより、部分冗長となる式のコード移動が可能となる範囲を広げ

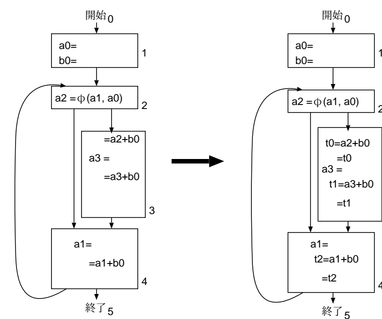

図 8: 従来の SSAPRE で最適化効果のないプログラムの例

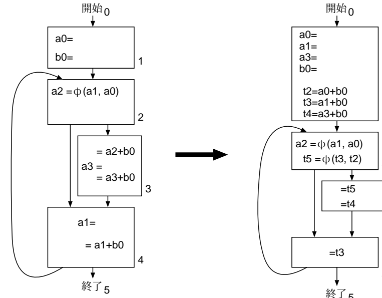

図 9: 図8に本手法を適用した結果

るものである。本手法は、New PRE 前処理部分と New PRE 本処理部分から構成されている。

New PRE 本処理部分のデータフロー解析は、Knoop らが提案した *lazy code motion* とよばれる部分冗長性除去のアルゴリズム [8] を基にしている。単純なコード移動では、可能な限りプログラム中の開始ブロックの近くに計算式を置くことで部分冗長性除去を実現できる。しかし、これではレジスタの生存区間 (*live range*) を長くしてしまうのでレジスタ圧力 (*register pressure*) が増え、ひいてはレジスタスピルを誘発し、プログラムの実行時間を遅くする。そこで *lazy code motion* では、最適化効果が同じであると見込めるプログラムポイントの中で、できるだけ遅い位置に計算式を移動し、レジスタの負担を軽減している。本研究の特徴の一つは、emphlazy code motion を自然な形で SSA 形式に適用する方法を示したことである。さらに本研究では、変更文に関してもレジスタ圧力を考慮したコード移動を行っている。

本手法では、最適化効果を高めるために本手法への入力形式にいくつかの制約を設ける。本手法への入力形式は、刈り込んだ SSA 形式となっているものとする。本手法のアルゴリズムは刈り込んだ SSA 形式に限らなくても正しく動作を行うが、刈り込んだ SSA 形式が最も変更文の移動可能となる範囲が大きくなるためである。

本節では、図10の制御フローグラフを例としてアルゴリズムの説明を行う。説明を簡単にするために、冗長となる計算式は $a + b$ の一種類のみとし、$a + b$ と記した場合には通常形式の $a + b$ ではなく SSA 形式の $a_1 + b_1$ や $a_3 + b_2$ などを全て含むものとする。式は三番地コードとし、演算子は $+$ に限定しているが一般性を失わない。三番地コードへの分解の仕方によって全ての共通部分式が検出されされ易いことがわるが、このことは以前より知られている [3]。

また、本手法では、字面上で代入文で定義された変数のみを用いた計算式を**純粋な計算式**と呼び、単に計算式

と呼ぶときは全ての計算式を対象とする。

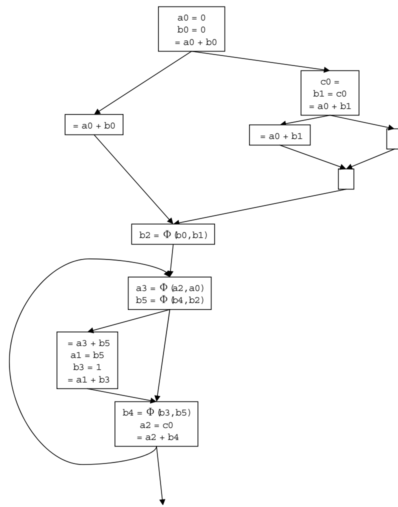

図 10: 例題

4.1 アルゴリズムの概要

1. 危険辺の除去
2. New PRE 前処理
   - 2 - a 計算式の ϕ 関数の作成
   - 2 - b 部分冗長となる式の候補作成
   - 2 - c 新しくできる ϕ 関数の添字の解析
3. New PRE 本処理
   - 3 - a 候補となる式に対するデータ依存解析
   - 3 - b 変更文の移動可能先の解析
   - 3 - c 計算式のコード移動先の解析
   - 3 - d 計算式のコード移動先の決定
   - 3 - e 変更文のコード移動先の決定

New PRE 本処理は、変更文の移動可能なプログラムポイントの解析と部分冗長な式の移動可能な場所の解析を行う。それぞれデータフロー方程式を解くことで求められる。データフロー方程式は基本ブロック内の局所的な性質を求め、ついで基本ブロック間の大域的な性質を解いていくことで求められる。

4.2 危険辺の除去

任意の制御フローグラフでは，コードの移動がその過程で**危険辺** (*critical edge*) によって阻まれることがある。危険辺とは図11(左) のブロック 3 からブロック 2 への辺のように，後続ブロックを複数持つブロックから先行ブロックを複数持つブロックへの辺のことである。

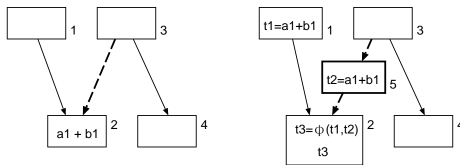

図 11: 危険辺の例とその除去

ブロック 2 にある $a+b$ を移動したい場合、ブロック 1 に移動しても 1 から 2 へのパスについてのプログラムの意味は変わらないが、ブロック 3 に計算を移動しようとすると、ブロック 3→ ブロック 4 へのパスに影響を与えてしまうため、この移動ができない。そこであらかじめ、**危険辺の除去** (*edge splitting*) をしておくことで、この問題を回避できる。具体的には、図11(右) のように危険辺上に空のブロック 5 を挿入する。これにより、ブロック 5 に計算を置くことで、ブロック 3→ ブロック 4 へのパスに影響を与えることなく、ブロック 2 の先行ブロックに計算を移動できるようになった。

図10に危険辺の除去を施した結果は図12のようになる。

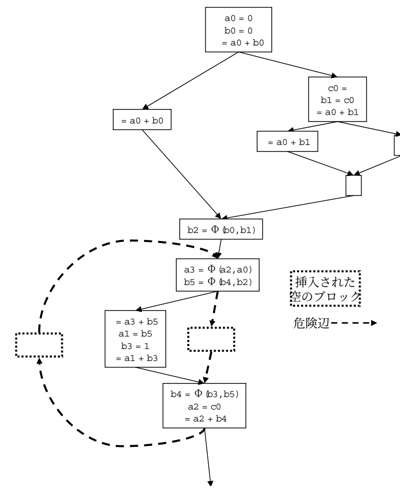

図 12: 危険辺除去の結果

4.3 New PRE 前処理

本節では、New PRE 前処理について説明をする。New PRE 前処理では、

1. 計算式の ϕ 関数の作成
2. 部分冗長となる式の候補作成
3. 新しくできる ϕ 関数の添字の解析

について解析を行う。

本手法では、変更文の移動を行うため、同一 ϕ 関数内の変数の生存区間の干渉を起こしてしまう。従来の PRE の手法 [9, 15] では、部分冗長性除去を行えないので、工夫をする必要がある。

本研究では、New PRE 前処理として、「計算式の ϕ 関数」というものを作成する。これは、コード挿入で $t=a+b$ を挿入するとき、同時にその後続ブロックで必要に応じて挿入する ϕ 関数の引数を決めるために必要となるものである (詳しくは4.3.1節で後述する)。このような ϕ 関数を作成することで、次のような解析結果も同時に得られる。

1. 部分冗長となる可能性のある式の候補
2. コード移動の際に挿入する ϕ 関数のテンポラリの添え字

この「計算式の ϕ 関数」という考え方を導入することで、PRE を SSA 形式に適用する際に困難であった ϕ 関数の添字の問題を [15] が解決された。この解析以降、それぞれの部分冗長となる可能性のある式の候補に対してデータ依存解析をし、SSA 形式でそれぞれの式ごとに部分冗長性除去を行っていくことで同一 ϕ 関数内の変数の生存区間の干渉に対処をしていく。なお、最終的に ϕ 関数内の生存区間に干渉が残っても Sreedhar らの SSA 逆変換 [13] を用いていればよいので問題はない。

4.3.1 計算式の ϕ 関数の作成

本節では、「計算式の ϕ 関数」の作成について説明をする。

計算式で実際に使用される変数の値は、代入文で定義された値のみである。しかし、ϕ 関数を越えた変数を用いた計算式は、実際に計算している値を一意に決めることができない。そのため、ϕ 関数で定義されている変数を用いた計算式が部分冗長であると認識したときには、その計算式の意味を変えないように ϕ 関数を入れる必要がある。

例えば、図13(左) での $a_2 + b_0$ で計算されている値は、$a_0 + b_0$ かもしれないし、$a_1 + b_0$ かもしれない。そのため、$a_0 + b_0$ や $a_1 + b_0$ で部分冗長となるとき、$t = a + b$ を挿入すると同時に、t の ϕ 関数を挿入する必要がでてくる。そこで、図13(中) のように $a_2 + b_0$ を左辺、$\phi(a_0 + b_0,\ a_1 + b_0)$ を右辺とする ϕ 関数を仮に作る。これにより、ϕ 関数の引数に現れるそれぞれの計算式に変数を割り当てることで、t の ϕ 関数を作ることができる。図13(右) では、$a_2 + b_0$ に t_2、$a_0 + b_0$ に t_0、$a_1 + b_0$ に t_1 を割り当てた後、コード移動を行っている。

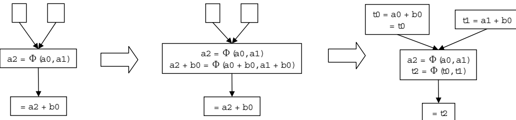

図 13: 計算式の ϕ 関数とコード移動した例

計算式の ϕ 関数を作るアルゴリズムは下記のようになる。

1. CFG の終了ブロックから有向辺を逆向きに辿りながら、計算式にぶつかったら、二つの変数がそれぞれ ϕ 関数で定義されたものなのか、代入文で定義されたものなのかを調べる
2. 計算式の 2 つの変数を調べる
   - 2 - a 2 つの変数が両方とも ϕ 関数で定義されていれば、二つの ϕ 関数を合成して新しい ϕ 関数を作る
   - 2 - b 一方が ϕ 関数、他方が代入文で定義されていれば、ϕ 関数に代入文で定義された変数を組み込んだ新しい ϕ 関数を作る
   - 2 - c 両方とも代入文で定義された式、もしくは、既に解析された式であればなにもしない (終了条件)
3. ϕ 関数が作成されたら、新しく作成された ϕ 関数の右辺の引数となっている計算式に対しても同様の解析を行う

以下、計算式の ϕ 関数の作成例として、図 14、15を示す。図 14では、2 - a の例、図15は、2 - b の適用後に 2 - a を適用した例である。

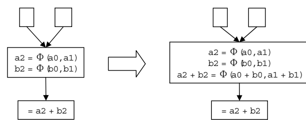

図 14: 2 つの変数が同一ブロックで ϕ 関数で定義されている場合

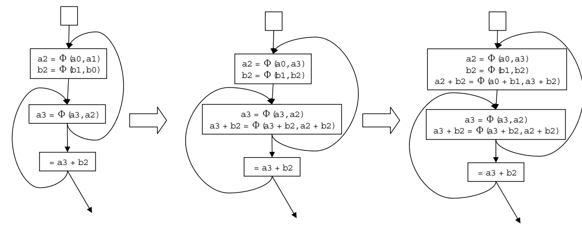

図 15: ループで ϕ 関数が定義されている場合

4.3.2 新しい ϕ 関数の添字の解析

部分冗長性除去では計算の挿入を行うとき、$t = a + b$ という形でテンポラリに値を代入する。その際、t の添え字に注意する必要がある。同時に t の後続パスで、必要に応じて ϕ 関数を挿入する必要がある。その際にも ϕ 関数の引数の添え字に注意をしなければならない。「計算式の ϕ 関数」の作成により、作成された新しい ϕ 関数の右辺と左辺の計算式に対して、固有の添え字をもつテンポラリを割り当てていく。割り当てられたテンポラリと計算式の対応表は $SuffixTable$ に記述しておけばよい。

例えば、図13では、新しい ϕ 関数 として $a_2 + b_0 = \phi(a_0 + b_0,\ a_1 + b_0)$ が作成された。このとき、テンポラリの添え字は $t_0 \to a_0 + b_0$、 $t_1 \to a_1 + b_0$、$t_2 \to a_2 + b_0$ となる。

4.3.3 部分冗長となる可能性のある式の候補の作成

求められた「計算式の ϕ 関数」の引数の中で純粋な計算式 (前述) が冗長となる可能性のある計算になる。これを部分冗長な式の候補とし、それぞれの候補に対して部分冗長性除去を行っていく。候補となる純粋な計算式は $CandidateTable$ に記述しておく。

例えば、図13で a_0、a_1、b_0 が代入文で定義された変数であるとすると、新しい ϕ 関数 の右辺の計算式である $a_0 + b_0$、$a_1 + b_0$ が部分冗長性除去の候補となる。

図 10 に New PRE 前処理を適用した結果は 図16のようになる。

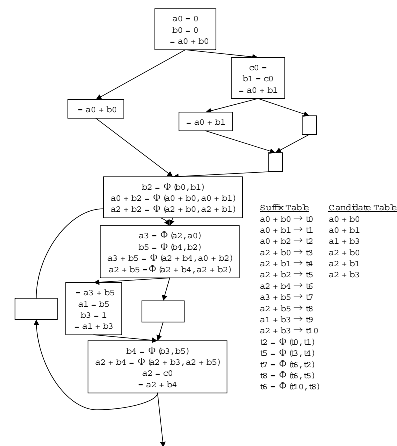

図 16: 前処理をした結果

4.4 New PRE 本処理

New PRE 前処理が終わったら New PRE 本処理をする。本節では、New PRE 本処理について説明をする。

4.5 局所的な性質

本節では、本アルゴリズムを通じて必要となる局所的な性質についての説明をする。これは Knoop の *optimal code motion*[9] で提案されている局所的な性質を SSA 形式に適用したものである。

基本ブロック内では、共通部分式の除去はされているものとする。そのため、基本ブロック内で $a+b$ の計算が連続して現れることはない。以下に、ひとつの基本ブロックのなかで、$a+b$ の計算に関係のある部分 (a や b の使用や、a や b の変更文) だけに注目した例を示す。

プログラム 1 (基本ブロックの内部)

$$a_2 = \phi(a_0, a_1); \quad (1)$$
$$b_2 = \phi(b_0, b_1); \quad (2)$$
$$c_2 = \phi(c_0, c_1); \quad (3)$$
$$x_1 = a_2 + b_2; \quad (4)$$
$$a_3 = c_2; \quad (5)$$
$$y_1 = a_3 + b_2; \quad (6)$$
$$b_3 = d_1; \quad (7)$$
$$z_1 = a_3 + b_3; \quad (8)$$

基本ブロック内を以下の 3 つの部分に分けて考えることにする (図17も参照)。まず， ϕ 関数の部分 (1〜3) を分けて考え、その部分を ϕ **関数部分**とよぶ。残りのうち他の基本ブロックの $a+b$ と冗長になる可能性がある計算は、 a や b の値を変更する文 (ϕ 関数を除く。以後、変更文とよぶ) より前にある最初の $a+b$ (4)(先行する基本ブロックの $a+b$ と冗長になる可能性がある) と、 a や b の変更文より後にある最後の $a+b$ (8) (後続する基本ブロックの $a+b$ と冗長になる可能性がある) だけである。その 2 つを分けた方が考えやすいので、基本ブロックを ϕ 関数の部分と残り 2 つの部分に分けて考えることにする。残り 2 つの部分の分け目は ϕ 関数以外の a や b の変更文の最後のもの (7) の直後とする。分けたものの最初の方を**入口部分**とよび、後の方を**出口部分**とよぶ。変更文がないときは、その基本ブロックの ϕ 関数以外を入口部分とし、出口部分は空とする。入口部分の a や b に代入する文より前にある $a+b$ の計算 (4) を**入口計算**とよび、出口部分の計算 (8) を**出口計算**とよぶ。 変更文を移動するとき、変更文の移動を妨げるのは、変更文の右辺を定義している代入文と変更文の右辺を定義している ϕ 関数である。このような代入文を**変更定義文**とよび、ϕ 関数を**変更** ϕ **関数**とよぶ。

本アルゴリズムでは、計算式や変更文を適当なところに挿入するが、その挿入点は各部分に 1 つずつ設ける。計算式の挿入点は、出口計算がなくて変更文があるときは変更文の直後、入口計算や出口計算があるときはその計算の直前、計算も変更文もなければそのブロックの最後とする。 変更文の挿入点は、変更定義文がある場合は出口計算の直前、それ以外のときは ϕ 関数部分の直後に挿入することにする。また、新しい変数の ϕ 関数の挿入点は、 ϕ 関数部分の最後の部分とする。例を図 17 に示す。

以上により求められる局所的な性質を次のように定義する。

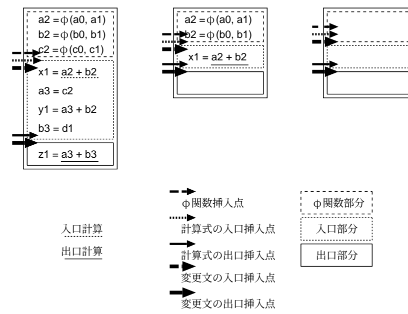

図 17: $a+b$ に関する基本ブロックの性質

局所的な性質 定義 1

$\mathcal{N}Comp(c,B)$	:	(ブロック)B に計算式 c についての入口計算がある (あれば $true$、なければ $false$ 以下同様)
$\mathcal{X}Comp(c,B)$	:	B に計算式 c についての出口計算がある
$Transp(c,B)$	:	B に計算式 c の変更文はない
$ModTransp(m,B)$	:	B に変更文 m の変更定義文はない
$ModPhiTransp(m,B)$	:	B に変更文 m の変更ϕ関数はない

4.5.1 データ依存解析

データ依存解析は、部分冗長な計算が行われている場所を調べるためのものである。 本研究がデータ依存解析により同一の計算がなされている箇所を探す理由は、変更文の移動をした際、同一 ϕ 関数で生存区間干渉を起こす可能性があるためである。

本研究ではまず、それぞれの候補の変数の推移的データ依存関係のある式を抽出する。その後、それぞれの変数で求められた集合の積集合を部分冗長性除去の対象としていく。$\mathcal{N}Comp$、 $\mathcal{X}Comp$ や $Transp$ のフラグ付けはここで行われる。

例えば、図18での a_0 にデータ依存する式は、(3, 4, 7, 8, 10, 12) である。また、b_0 にデータ依存する式は、(3, 4, 9, 11, 12, 16, 18) である。この積集合は (3, 4, 12) となり、a_0+b_0 の計算が実際に計算される箇所が抽出された。

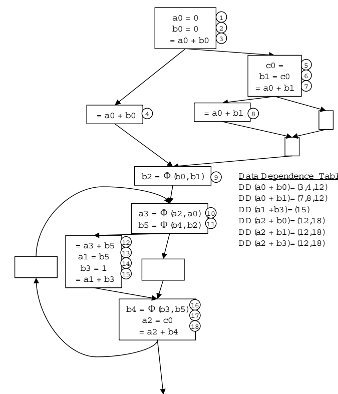

図 18: データ依存解析をした結果

4.5.2 大域的な性質

以後、様々な大域的な性質を定義することになるが、説明の便宜上、頭文字に $\mathcal{N}$(入口部分の意味) や $\mathcal{X}$(出口部分の意味) をつけて定義したものを $\mathcal{N}$ や $\mathcal{X}$ を付けずに説明に用いるときがある。これは、$\mathcal{N}$ や $\mathcal{X}$ の区別をする必要がない場合で、 $\mathcal{N}$ と $\mathcal{X}$ 両方を指している。

ここで $t=a+b$ を挿入すべき場所を $Insert$、$a+b$ の計

算を t に置き換えるべき場所を $Replace$ とする。また、t の定義文を挿入することで、t の後続パスで ϕ 関数を必要に応じて挿入する必要がある。ϕ 関数を入れるべき場所を $PhiInsert$ とする。変更文を入れるべき場所は $ModInsert$ とする。それぞれは、次のように定義できる。

大域的な性質 定義 1

$\mathcal{N}Insert(c, B)$	:	B の入口挿入点に計算 $t = c$ を挿入すべきである.
$\mathcal{X}Insert(c, B)$	:	B の出口挿入点に計算 $t = c$ を挿入すべきである.
$\mathcal{N}Replace(c, B)$	:	B の入口計算 c を t で置き換えるべきである。
$\mathcal{X}Replace(c, B)$	:	B の出口計算 c を t で置き換えるべきである。
$PhiInsert(p, B)$	:	B のϕ関数挿入点に t のϕ関数 p を挿入すべきである。
$\mathcal{N}ModInsert(m, B)$	:	B の入口挿入点に変更文 m を挿入すべきである。
$\mathcal{X}ModInsert(m, B)$	:	B の出口挿入点に変更文 m を挿入すべきである。

これらは大域的な性質であり、前に求めた局所的な性質や、それらを利用して求めた別の大域的な性質を利用しながら制御フローグラフを解析して求める。

4.5.3 変更文を移動可能なプログラムポイントの解析

この節では、変更文を上方に移動可能なプログラムポイントの解析方法について述べる。変更文を移動する際、以下のような点を考慮に入れる必要がある。

1. 変更文の右辺で使われている変数の定義を越えた変更文のコード移動はできない
2. SSA 形式を崩さない
   SSA 形式上で定義されている変数はその変数の使用を支配している必要があるため (Dominance Property [2])、その変数の定義がなされているブロックの支配木の親になるブロックにのみ移動可能である。

以上の条件を満たす変更文の移動可能な場所を求めるために、まず条件 1 を満たす場所を求める。その後、条件 2 も同時に満たす場所を解析する。

変更文の移動可能な場所は以下のように定義できる。また、変更文を移動することができる場所のなかで、できるだけ開始ブロックに近く移動させることのできるプログラムポイントは次のように定義できる。

大域的な性質 定義 2

$\mathcal{N}MayMove(m, B)$	:	B の入口挿入点において、変更文 m の右辺の定義文をまたいではいないが、Dominance Propety を守っていないかもしれない
$\mathcal{X}MayMove(m, B)$	:	B の出口挿入点において、変更文 m の右辺の定義文をまたいではいないが、Dominance Propety を守っていないかもしれない
$\mathcal{N}CanMove(m, B)$	:	B の入口挿入点に変更文 m を移動することができる
$\mathcal{X}CanMove(m, B)$	:	B の出口挿入点に変更文 m を移動することができる
$\mathcal{N}ModEarliest(m, B)$	:	B の入口挿入点に変更文 m を移動することができるが、それ以上開始ブロックに近いところに移動させることはできない
$\mathcal{X}ModEarliest(m, B)$	:	B の出口挿入点に変更文 m を移動することができるが、それ以上開始ブロックに近いところに移動させることはできない

$\mathcal{N}MayMove$ と $\mathcal{X}MayMove$ は、次のような方程式を解くことで求められる。

データフロー方程式 1 (MayMove)

$$\mathcal{N}MayMove(B) = \begin{cases} \mathcal{X}MayMove(B) \cap ModTransp(B) & \textbf{\textit{if}}\ B = \text{開始} \\ true & \textbf{\textit{otherwise}} \end{cases}$$

$$\mathcal{X}MayMove(B) = \bigcap_{S \in succ(B)} (ModPhiTransp(S) \cap \mathcal{N}MayMove(S))$$

これらの式の右辺は、それぞれ「ブロック B の出口挿入点に変更文の移動が可能で、かつ、出口挿入点までに変更定義文が存在しない」、「ブロック B の後続ブロック全てにおいて、入口挿入点に変更文の移動が可能で、かつ、変更 ϕ 関数が存在しない」ことを意味する。

次に、条件 2 である Dominance Property を守るために、求められた $\mathcal{N}MayMove$ と $\mathcal{X}MayMove$(条件 1) を支配木とマージさせることにより、それぞれ、$\mathcal{N}CanMove$ と $\mathcal{X}CanMove$ を求めることができる。ここでいうマージとは、求めたい変更文のあるブロックから、支配木上で親となるブロックと $MayMove$ と $CanMove$ の結果を、 図 19 と図20に示す。

例えば、変更文 a_2 は、b_2 の ϕ 関数による定義文の直後まで移動可能である。

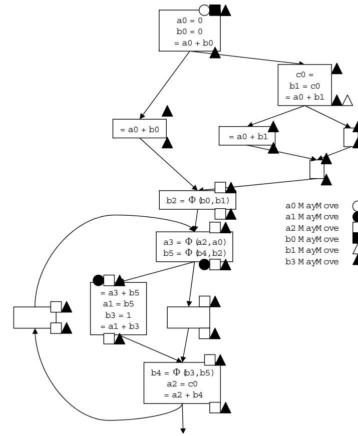

図 19: MayMove を求めた結果

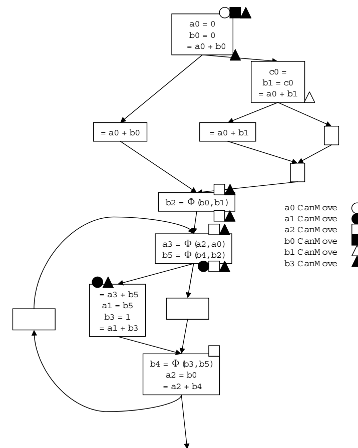

図 20: CanMove を求めた結果

求められた $CanMove$ のなかで、支配木上で親となるブロックがプログラムの開始ブロックにより近いブロックとなる。$ModEarliest$ は、支配木でできる限り親となるブロックを辿ったものとなる。$ModEarliest$ の結果は図21のようになる。

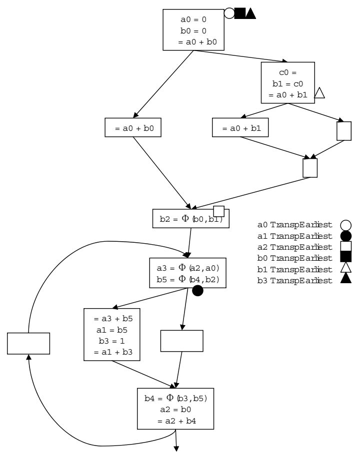

図 21: ModEarliest を求めた結果 (求めたブロックに移動することにより示してある)

4.5.4 Earliest

変更文が移動されることを前提として、計算式をコード移動しても安全である範囲で、できるだけ開始ブロックの近くまで移動できる場所を求める。

大域的な性質 定義 3

$\mathcal{N}Earliest(c, B)$: B の入口挿入点に計算式 c を挿入することができるが、それ以上開始ブロックに近いところに移動させることはできない

$\mathcal{X}Earliest(c, B)$: B の出口挿入点に計算式 c を挿入することができるが、それ以上開始ブロックに近いところに移動させることはできない

SSA 形式の性質上、計算式は変更文が支配する場所に位置していなければならない。そのため、計算式が移動できる範囲は、全ての計算式の変更文の *ModEarliest* となる場所が同時に支配する場所となる。そのなかで、支配木上でなるべく親になる場所が *Earliest* となる。求められた *Earliest* のうち、変更文のある場所が同じブロックであるところは、 $\mathcal{X}Earliest$、そうでなければ、 $\mathcal{N}Earliest$ となる。*Earliest* を解析した結果は、図22となる。

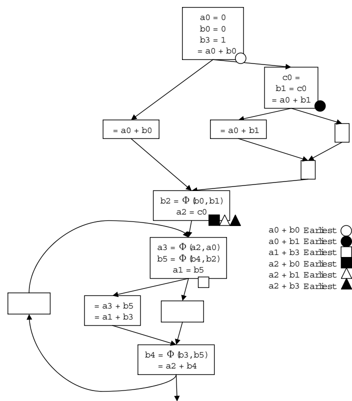

図 22: Earliest を求めた結果

4.5.5 Delay

Earliest で計算式をプログラムの意味が変わらない範囲で出来るだけ引き上げる場所を求めたが、むやみに計算式の引き上げを行うとレジスタ圧力が上り、プログラムの実行速度を落としてしまう。*Delay* は、プログラムの意味の変わらない範囲で式の挿入点を遅くし、レジスタ圧力を下げる。

大域的な性質 定義 4

$\mathcal{N}Delay(c, B)$: プログラムの入口から B の入口挿入点に達するどのパスにも、$\mathcal{N}Earliest(P)$ または $\mathcal{X}Earliest(P)$ である挿入点があり、かつ、その計算式 c を B の入口挿入点まで遅れさせてもよい。

$\mathcal{X}Delay(c, B)$: プログラムの入口から B の出口挿入点に達するどのパスにも、$\mathcal{N}Earliest(P)$ または $\mathcal{X}Earliest(P)$ である挿入点があり、かつ、その計算式 c を B の出口挿入点まで遅れさせてもよい。

データフロー方程式 2 (Delay)

$$\mathcal{N}Delay(c, B) = \mathcal{N}Earliest(c, B) \cup \begin{cases} false & \textbf{\textit{if}}\ B = \text{開始} \\ \bigcap_{P \in pred(B)} \{\mathcal{X}Delay(c, P) \cap \overline{\mathcal{X}Comp(c, P)}\} & \textbf{\textit{otherwise}} \end{cases}$$

$$\mathcal{X}Delay(c, B) = \mathcal{X}Earliest(c, B) \cup \{\mathcal{N}Delay(c, B) \cap \overline{\mathcal{N}Comp(c, B)}\}$$

これらの式の右辺は、それぞれ、「ブロック B の入口挿入点に計算式 c を遅れさせられるのは、ブロック B の入口挿入点で *Earliest* であるか、もしくは、全ての先行ブロックの出口部分に計算を遅れさせられてかつ出口計算がない場合である。ただし、開始ブロックには遅れさせることは出来ない」、「ブロック B の出口挿入点に計算式 c を遅れさせられるのは、ブロック B の出口挿入点で *Earliest* であるか、もしくは、入口部分に計算式を遅れさせられてかつ入口計算がない場合である」ことを意味する。*Delay* を求めた結果は図23のようになる。

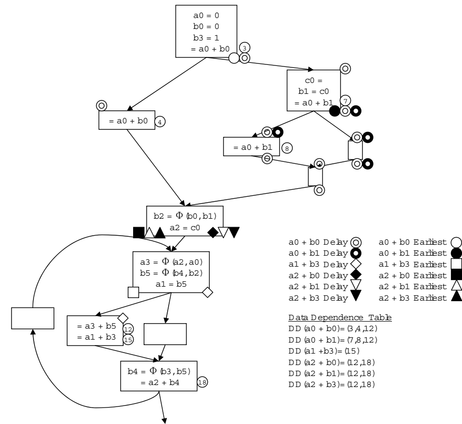

図 23: Delay を求めた結果

4.5.6 ReMakeSuffix

計算式のコード移動をするとき、必要に応じて ϕ 関数を挿入する必要がある。必要となる ϕ 関数は New PRE 前処理で求めた「計算式の ϕ 関数」を用いることで求められる。

しかし、計算式の ϕ 関数を挿入する際、計算式の ϕ 関数の右辺の計算式の *Delay* の結果次第で、計算式の ϕ 関数の代わりに左辺の計算式を挿入すればよい場合がある。

以下、説明の簡単化のために ϕ 関数の引数は二つであるとするが、一般性を失うことはない。

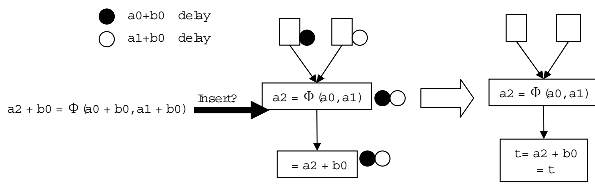

図 24: 合流ブロックで計算式の ϕ 関数を挿入する代わりに計算式の ϕ 関数の左辺の計算式でを挿入する例

例えば、図24では、合流ブロックで計算式の ϕ 関数 $a_2 + b_0 = \phi(a_0 + b_0, a_1 + b_0)$ が挿入される可能性がある。部分冗長となる計算式の候補である $a_1 + b_0$ と $a_0 + b_0$ の *Delay* は 計算式の ϕ 関数を挿入するブロックで同時に *true* であると仮定する。すると、合流ブロックの先行ブロック以前で $t_0 = a_0 + b_0$ と $t_1 = a_1 + b_0$ を挿入することはなくなったので、これらの計算式のコード移動に伴う ϕ 関数は挿入する必要がなくなる。従って、例での $a_0 + b_0$ や $a_1 + b_0$ の計算式が、計算式の ϕ 関数の挿入するブロック以降で挿入される可能性があるという場合、この計算式の ϕ 関数を挿入する必要はないことがわかる。

また、この例の場合、$a_0 + b_0$ と $a_1 + b_0$ は、計算式の ϕ 関数の左辺で置き換えることが出来る。図24での $a_2 + b_0$ は、$a_0 + b_0$ と $a_1 + b_0$ を網羅する。よって、$a_0 + b_0$ と $a_1 + b_0$ の *Delay* が同時に *true* なのであれば、これらの計算式を $a_2 + b_0$ に置き換えた方が得である。これによって、レジスタ圧力と計算回数を同時に改善できる。

さらに、このとき、新たに $a_2 + b_0$ の計算式を用いることになったので、その計算式の *Delay* を求めることでレジスタ圧力の軽減を再び試みることが出来る。

計算式の ϕ 関数とその ϕ 関数の右辺の計算式の *Delay* の位置関係には以下のような場合が考えられる。

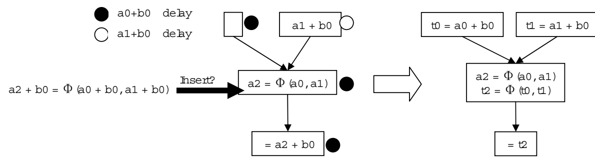

図 25: 合流ブロックで計算式の ϕ 関数を挿入する例

1. 計算式の ϕ 関数の挿入する場所が支配しているブロックにおいて、右辺の計算式のうち、片方の計算式、もしくは両方とも *Delay* することができない。このとき、その *PhiInsert* を true にする。
   例えば、図25では、$a_1 + b_0$ の *Delay* が、計算式の挿入ブロックの先行ブロックで止まっている。この場合、計算式の ϕ 関数を挿入する代わりに左辺の計算式を挿入してしまうと部分冗長性除去が行えない。どのパスを通っても $a + b$ の計算を一回で済ますためには、左の先行ブロックに $a_0 + b_0$ の計算式を挿入し、計算式の ϕ 関数を t の ϕ 関数に直したものを挿入することで解決できる。ただし、実際の計算式のコード挿入は、以後の *Latest* の結果を用いるので、ここまでの解析で計算式の挿入場所は分からない。この解析によって分かることは、「計算式の ϕ 関数をテンポラリの ϕ 関数に直したものを挿入すること」と「右辺の計算式がどこかに挿入されること」である。
   同様に、図27は、計算式がループ文の外まで *Delay* 出来る例である。 この場合も、計算式の ϕ 関数の右辺の計算式が、その計算式の ϕ 関数のブロックが支配する場所に *Delay* 出来ないため、計算式の ϕ 関数の左辺で置き換えることは出来ない。これは、計算式の ϕ 関数である $a_2 + b_0 = \phi(a_1 + b_0, a_0 + b_0)$ に対応するテンポラリの ϕ 関数を挿入することにし、右辺の計算式である $a_1 + b_0$ と $a_0 + b_0$ をループ文の前に挿入することで解決する。
2. 計算式の ϕ 関数の挿入場所が支配しているブロックに、計算式の ϕ 関数の右辺の計算式を全て *Delay* できる場合。このとき、計算式の ϕ 関数を挿入する代わりに、新たに計算式の ϕ 関数の左辺を計算式とみなし、その計算式の *Delay* を求める。新たな計算式の *Delay* が *true* となる場所は、計算式の ϕ 関数の右辺の計算式が全て *true* となる場所である。
   図24については、既に上記で説明をしているので省略をする。図26では、ループのなかの変更文が計算式のコード移動を妨げている例である。このときも、$a_0 + b_0$ や $a1 + b0$ の計算式を個別に挿入する代わりに、ϕ 関数の左辺である $a_2 + b_0$ を挿入することで不必要な計算を回避している。

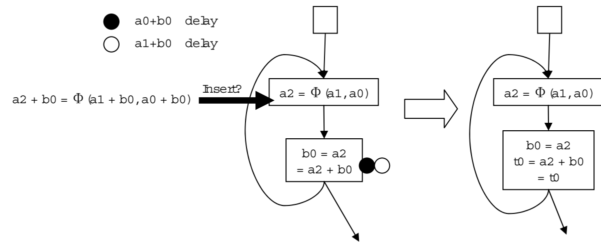

図 26: ループ文で計算式の ϕ 関数挿入する代わりに計算式の ϕ 関数の左辺の計算式を挿入する例

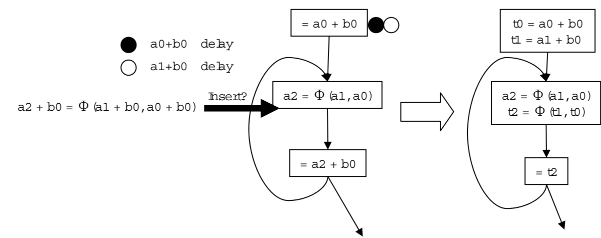

図 27: ループ文で計算式の ϕ 関数を挿入する例

ここで局所的な性質 *ReMakeSuffix* を次のように定義する。また、計算式の ϕ 関数 P は $p_2 = \phi(p_0, p_1)$ とする。

局所的な性質 定義 2 (ReMakeSuffix)

$ReMakeSuffix(p_2, B)$: ブロック B に挿入する計算式のϕ関数 P を挿入する代わりに、ϕ関数の右辺である p_2に置き換えるべきである。

$ReMakeSuffix(p, B)$ は、ϕ 関数 P が $p_2 = \phi(p_0, p_1)$ のとき、次の式で計算することが出来る。

データフロー方程式 3 (ReMakeSuffix)

$$ReMakeSuffix(p_2, B) = (\mathcal{N}Delay(p_0, B) \cap \mathcal{N}Delay(p_1, B)) \\ \cup(\mathcal{X}Delay(p_0, B) \cap \mathcal{X}Delay(p_1, B))$$

この式は、「計算式の ϕ 関数 P を挿入する代わりに計算式の ϕ 関数の左辺の計算式で置き換えられるのは、ブロック B の入口挿入点で計算式の ϕ 関数の右辺の計算式 p_0 と p_1 がともに *Delay* である場合か、ブロック B の出口挿入点で p_0 と p_1 がともに *Delay* である場合である」という意味である。

この計算から、挿入する可能性のあった ϕ 関数の左辺の計算式を挿入することが解析されれば、その計算式の *Delay* を求めることになる。その新たな計算式 p_2 の *Delay* は次のような式によって求められる。

データフロー方程式 4 (Delay)

$$\mathcal{N}Delay(p_2, B) = \mathcal{N}Delay(p_0, B) \cap \mathcal{N}Delay(p_1, B)$$
$$\mathcal{X}Delay(p_2, B) = \mathcal{X}Delay(p_0, B) \cap \mathcal{X}Delay(p_1, B)$$

この式は、それぞれ、「ブロック B の入口部分において、新たな計算式 p_2 を遅れさせることが出来るのは、計算式 p_0 と p_1 が共に、ブロック B の入口部分において遅れさせることが出来る場合である」、「ブロック B の出口部分において、新たな計算式 p_2 を遅れさせることが出来るのは、計算式 p_0 と p_1 が共に、ブロック B の出口部分において遅れさせることが出来る場合である」という意味である。

その計算式の *Delay* によってはさらに他の 計算式の ϕ 関数を挿入する代わりに計算式を挿入することが出来る場合があり、これらを繰り返すことにより、レジスタ圧力の軽減を試みることが出来る。

図28に *ReMakeSuffix* をした結果を示す。

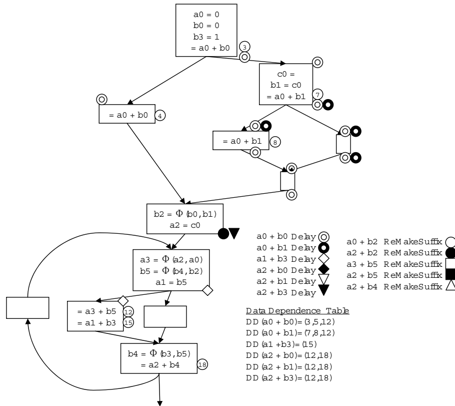

図 28: ReMakeSuffix を求めた結果

4.5.7 Latest

計算式の *Delay* の結果を使って、*Earliest* である場所から遅れさせていって、それ以上遅れさせることのできない場所を求めることで、最終的な計算式の挿入点が求められる。そのようなような場所 (*Latest*) を次のように定義する。

大域的な性質 定義 5

$\mathcal{N}Latest(c, B)$: B の入口挿入点で計算式 c が $\mathcal{N}Delay$であり、
それ以上出口ブロックに近いところに移動させることはできない。
$\mathcal{X}Latest(c, B)$: B の出口挿入点で計算式 c が $\mathcal{X}Delay$であり、
それ以上出口ブロックに近いところに移動させることはできない。

Latest は次のような式で求められる。

データフロー方程式 5 (Latest)

$$\mathcal{N}Latest(c, B) = \mathcal{N}Delay(c, B) \cap \mathcal{N}Comp(c, B)$$
$$\mathcal{X}Latest(c, B) = \mathcal{X}Delay(c, B) \cap \{\mathcal{X}Comp(c, B) \cup \bigcup_{S \in succ(B)} \overline{\mathcal{N}Delay(c, S)}\}$$

これらの式の右辺は、それぞれ、「ブロック B の入口部分において、計算式 c をブロック B の入口部分まで遅れさせることができ、かつ入口計算がある」、「ブロック B の出口部分まで遅れさせることができ、かつ出口計算があるかブロック B のいずれかの後続ブロックの入口部分に遅れさせることが出来ない」という意味である。*Latest* を求めた結果を図29に示す。

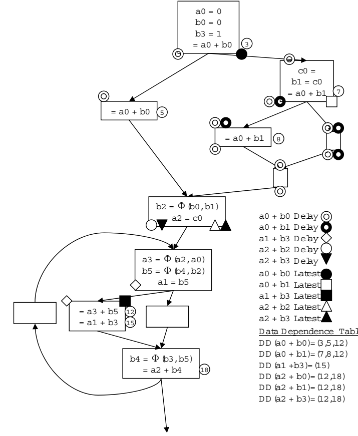

図 29: Latest を求めた結果

4.5.8 計算式のコード移動

Latest まで求め終わったら、この時点でコード移動を正しく行うことができる。この生存区間を短縮した地点へのコード移動を行う場合は、 それぞれの候補に対して求めた $\mathcal{N}Latest(B)$ が *true* である B の入口挿入点と $\mathcal{X}Latest(B)$ が *true* である B の出口挿入点に計算式 c のコード移動を行う。次に、データ依存解析で求めた $\mathcal{N}Comp(B)$ が *true* である B の入口計算と $\mathcal{X}Comp(B)$ が *true* である B の出口計算を、New PRE 前処理で求めた SuffixTable を参照して、対応する変数に置き換えればよい。すなわち、

$$\begin{aligned}
\mathcal{N}Insert(c, B) &= \mathcal{N}Latest(c, B) \\
\mathcal{X}Insert(c, B) &= \mathcal{X}Latest(c, B) \\
\mathcal{N}Replace(c, B) &= \mathcal{N}Comp(c, B) \\
\mathcal{X}Replace(c, B) &= \mathcal{X}Comp(c, B)
\end{aligned}$$

としてコード移動を行えばよい。計算式のコード移動をした結果

4.5.9 ModLatest

計算式の *Latest* まで求まったら、変更文もできるだけ出口ブロックの方向へコード移動したい。変更文を出来るだけ出口ブロックの方向へ移動させるためには、変更文の移動可能先を解析する際に求めた *CanMove*(変更文の移動可能な場所) を開始ブロックから辿りながら、その変更文を使用する全ての計算式を同時に支配するという条件

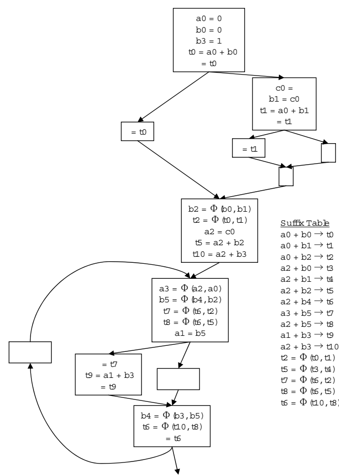

図 30: 計算式のコード移動をした結果

を満たす場所を求めていけばよい。この条件を初めて満たさない場所の直前の条件を満たす場所が $ModLatest$ が $true$ となる。$CanMove$ を辿ることによって $Dominance\ Property$(変更文がそれを使用する計算式を支配すること) は満たしている。

例えば、図31の変更文 b_3 が使用される場所は t_{10} の計算式と t_9 の計算式と b_4 の ϕ 関数である。これらを全て同時に支配し、かつ、b_3 の $CanMove$ が $true$ である場所は、b_2 の ϕ 関数のあるブロックであることがわかる。

$ModLatest$ は次のように定義できる。

大域的な性質 定義 6 (ModLatest)

$\mathcal{N}ModLatest(m,B)$: $ModEarliest$から B の入口挿入点の間に $\mathcal{N}Latest$も $\mathcal{X}Latest$もなく、それ以上出口ブロックに近いところに変更文 m を移動させることはできない。

$\mathcal{X}ModLatest(m,B)$: $ModEarliest$から B の出口挿入点の間に $\mathcal{N}Latest$も $\mathcal{X}Latest$もなく、それ以上出口ブロックに近いところに変更文 m を移動させることはできない。

$ModLatest$ を求めた結果は図31となる。

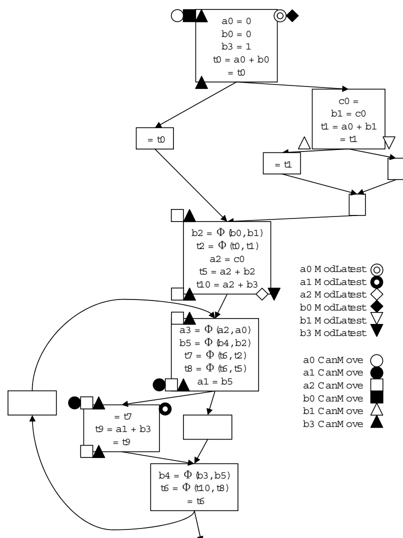

図 31: ModLatest を求めた結果

4.5.10 変更文のコード移動

$ModLatest$ まで求まったら、変更文のコード移動を行う。変更文のコード移動では、$\mathcal{N}ModLatest(m,B)$ が $true$ であるブロック B の入口挿入点と $\mathcal{X}ModLatest(m,B)$ が $true$ であるブロック B の出口挿入点に変更文 m のコード移動を行う。すなわち、

$$
\begin{aligned}
\mathcal{N}ModInsert(m,B) &= \mathcal{N}ModLatest(m,B) \\
\mathcal{X}ModInsert(m,B) &= \mathcal{X}ModLatest(m,B)
\end{aligned}
$$

のようにコード移動を行えばよい。

変更文のコード移動まで行った例を 図32に示す。

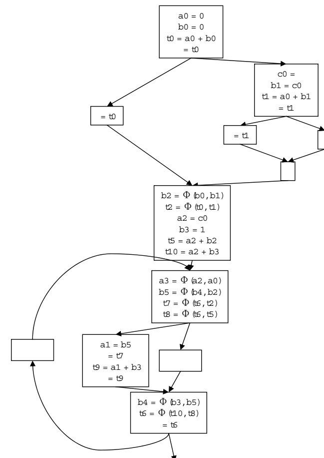

図 32: 変更文のコード移動をした結果

5 実装・評価

本研究では、COIN コンパイラ・インフラストラクチャ [11] の 1.0.2 版の SSA 最適化フェーズを用いて実装・評価した。COINS は Java 言語で実装されているため、本研究も Java 言語を用いて実装を行った。

5.1 実験

実験の目的および実験内容について説明をする。

5.1.1 実験環境

実験には SUN の Sun-Blade-1000 を用いた。主な仕様は表1のとおりである。

アーキテクチャ	Superscalar SPARC V9
プロセッサ	UltraSPARC-III 750MHz x 2
L1 キャッシュ	64KB(データ), 32KB(インストラクション)
L2 キャッシュ	8MB 外部キャッシュ
メモリ	1GB
OS	SunOS 5.8

表 1: Sun-Blade-1000 の主な仕様

5.1.2 実験の目的

本手法により、SSA 形式上の部分冗長性除去が見通しよく実装できる。また、変更文の移動を可能にすることで、より最適な場所への計算式のコード移動が可能にな

- 14queen
- Heap Sort
- Insertion Sort
- Selection Sort
- SPECint 164.gzip
- SPECint 181.mcf

表 2: 対象プログラム

る。本実験の目的は、本手法による最適化により実行時間がどれほど改善されるかを調べ、本手法の有用性を確かめることである。このために、以下のそれぞれの点を考慮した実験を行った。

1. 本手法が最適化アルゴリズムそのものとしての有用性
2. 変更文を移動することによる最適化効果の変化

以下、それぞれの実験について説明をしていく。

5.2 実験内容

それぞれの実験で計測をしたベンチマークプログラムは表2のものである。

5.2.1 前提条件

本手法による効果を計測するにあたり、前提条件となっている、SSA 変換、SSA 逆変換、危険辺の除去、そして三番地コードへの変換は全て COINS の SSA 最適化フェーズで実装されている。SSA 変換は刈り込んだ SSA 形式へ変換するものを、SSA 逆変換は Sreedhar らのアルゴリズム [13] を用いた。

5.2.2 実験1

実験1では、本手法の最適化アルゴリズムそのものとしての有用性を検証するため、COINS の SSA 最適化フェーズで実装されている他の最適化と、本手法による最適化の実行時間の計測を行った。他の最適化と本手法による最適化については表3に示す。

それぞれの最適化の組合せについて説明をする。実験1では、本手法が最適化そのもの効果としてどれほどの効果をもつのかを調べたい。OPT_0 は、最適化を何もしていないものとしての基準となる。OPT_1 は、プログラム最適化の中でも素朴な最適化をかけた場合である。

OPT_2 では、ループ不変式のループ外移動 ＋ 共通部分式除去を行う。この組み合わせは、部分冗長性除去とよく似た効果を持っており、ひとつの比較材料として採用した。OPT_2 は NEW_1 と比較するための最適化で、OPT_2 で最適化効果のあるプログラムでは NEW_1 も最適化効果を出してほしいという期待がある。

OPT_3 は、OPT_1 と OPT_2 を合わせたもので、OPT_1 と NEW_1 を合わせたものに当たる NEW_2 と比較するための最適化である。

また、本手法では、変更文の移動を可能としているため、局所的に生存変数の数が過多になる可能性が考えられる。本手法の有用性を検証するにあたり、静的なロードカウントも計測した。静的なカウントであるため、実行時のロード数による実効速度改善率を測れたわけではないが、変更文の移動によるレジスタ圧力の増加と計算回数の減少のトレードオフのひとつの指標となると考えた。

OPT_0 : SSA 変換 → SSA 逆変換
OPT_1 : SSA 変換
→ コピー伝播 → 定数伝播
→ 無用命令の除去 → SSA 逆変換
OPT_2 : SSA 変換
→ ループ不変式のループ外移動
→ 共通部分式の除去 → 無用命令の除去
→ SSA 逆変換
OPT_3 : SSA 変換
→ コピー伝播 → ループ不変式のループ外移動
→ 共通部分式の除去 → コピー伝播
→ 定数伝播 → 無用命令の除去
→ SSA 逆変換
NEW_1 : SSA 変換
→ 危険辺の除去 → 三番地コードへの変換
→ New PRE → 空ブロックの除去
→ 無用命令の除去 → SSA 逆変換
NEW_2 : SSA 変換
→ 危険辺の除去 → 三番地コードへの変換
→ 無用命令の除去 → コピー伝播
→ New PRE → コピー伝播
→ 定数伝播 → 空ブロックの除去
→ 無効命令の除去 → SSA 逆変換

表 3: 本手法と他の最適化の内様

5.2.3 実験2

実験2 では、変更文の移動によって部分冗長性除去の最適化効果がどれほど変わるかを調べたい。これは変更文を移動することを可能にした部分冗長性除去 (本手法) と変更文の移動をしない部分冗長性除去 (従来法) を比べることにより検証が可能である。

従来法としては、立川の方法 [15] を用いた。立川の方法は、一時変数に対する ϕ 関数の扱いが大変複雑であるが、効果としては Knoop のアルゴリズム [9] を SSA 形式用に適用したアルゴリズムである。これは、本研究における変更文を移動しないアルゴリズムとして位置する手法である。

本手法と従来法の実験については表4に示す。従来法における合併には Chaitin のアルゴリズム [4] を用いられた。NEW_1 と OLD_1、NEW_2 と OLD_2 はそれぞれの手法の前後で同等の処理を施しているので、それぞれの実行時間の改善率を比較することで、変更文移動による最適化効果の違いを検証できる。

6 実験結果

本節では、実験1と実験2 の実験結果を示す

6.1 実験1の結果

実行時間の計測には、表2の対象プログラムに対して、表3のそれぞれの最適化を適用した。実行時間の結果を表5に示す。表5を元に、最適化 0 の実行時間を基準とした実行時間の割合を表6に示す。図33は表6をグラフ化したものである。

OPT$_0$ ：SSA 変換 → SSA 逆変換
NEW$_1$ ：SSA 変換
→ 危険辺の除去 → 三番地コードへの変換
→ 本手法による部分冗長性除去 → 空ブロックの除去
→ 無用命令の除去 → SSA 逆変換
NEW$_2$ ：SSA 変換
→ 危険辺の除去 → 三番地コードへの変換
→ 無用命令の除去 → コピー伝播
→ 本手法による部分冗長性除去 → コピー伝播
→ 定数伝播 → 空ブロックの除去
→ 無効命令の除去 → SSA 逆変換
OLD$_1$ ：SSA 変換
→ 危険辺の除去 → 従来法
→ SSA 逆変換 → 合併
OLD$_2$ ：SSA 変換
→ 危険辺の除去 → 従来法
→ 定数伝播 → 無用コードの除去
→ SSA 逆変換 → 合併

表 4: 本手法と従来法の内様

	OPT$_0$	OPT$_1$	OPT$_2$	OPT$_3$	NEW$_1$	NEW$_2$
181.mcf	543	500	479	496	493	478
164.gzip	973	869	928	1011		950
14queen	99.48	99.47	92.15	85.13	91.24	85.28
Heap Sort	82.18	81.56	78.56	78.18	77.01	75.94
Insertion Sort	277.42	277.13	275.62	236.02	274.67	224.81
Selection Sort	359.66	361.04	339.39	347.51	324.10	361.91
tpprime	353.99	353.31	345.49	342.45	346.10	340.61

表 5: 実行時間 (sec)

プログラム	$\frac{OPT_0}{OPT_0}$	$\frac{OPT_1}{OPT_0}$	$\frac{OPT_2}{OPT_0}$	$\frac{OPT_3}{OPT_0}$	$\frac{NEW_1}{OPT_0}$	$\frac{NEW_2}{OPT_0}$
181.mcf	1.00	92.08	88.21	91.34	90.79	88.03
164.gzip	1.00	89.31	95.38	103.91		97.63
14queen	1.00	99.99	92.63	85.58	91.72	85.73
Heap Sort	1.00	99.24	95.59	95.13	93.70	92.40
Insertion Sort	1.00	99.90	99.35	85.09	99.01	81.04
Selection Sort	1.00	100.38	94.36	96.62	90.11	100.63
tpprime	1.00	99.81	97.60	96.74	97.77	99.22

表 6: 実行時間の割合 (%)

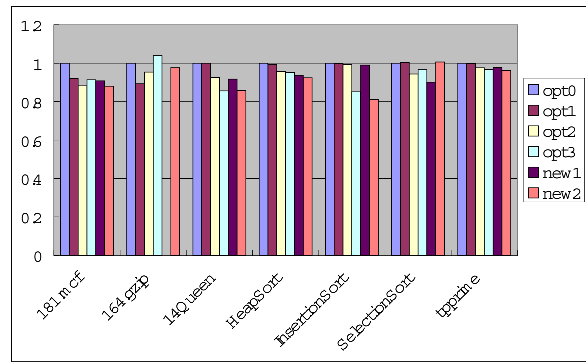

図 33: 本手法と他の最適化の実行時間の相対比

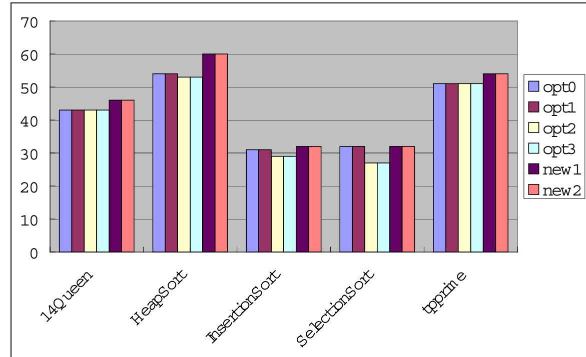

図 34: ロード命令数

6.2 実験1の考察

全体的に実行時間の結果を見渡してみて、OPT$_0$ を基準とした本手法の実行時間の割合は選択ソートを除いて100% 未満であり、最適化効果を示すことが出来たと言える。他の最適化と比べてみると、OPT$_2$ とほぼ同等もしくは最適化効果が大きいものであると言える。

14queen、Insertion Sort や 181.mcf での NEW$_2$ は、OPT$_2$ とほぼ同等の最適化効果を示せたといえる。特に Insertion Sort では本手法による最適化が強力な効果を示せている。

Heap Sort や tpprime では、OPT$_2$ よりも OPT$_3$、NEW$_1$ よりも NEW$_2$ で最適化効果の出ている例である。これは、最適化前処理として、コピー伝播などをすることで部分冗長となる式が増えた場合で、これによって最適化効果を大きくしたと考えられる。

図33全てのプログラムで共通して、本手法のロードの *static count* は最適化前よりも上昇している。ロードの数が増える理由は、変数の生存区間干渉が増えたため、レジスタ圧力が高まり、レジスタに乗り切らないため、一時的にメモリに退避させるため (スピル) だと考えられる。本手法で変数の生存区間干渉が増える理由は、

1. 計算式の挿入による変数の増加
2. 他の変数の生存区間への変更文の移動

が考えられる。 6.5節でこれらの問題について、それぞれ具体的な状況とともに説明する。

6.3 実験2の結果

従来法の実験 [15] でも本研究と同様に最適化 0 を計測している。本手法と従来法を比較するためには実際に従

来法を同じ COINS の版で実装し、同じ環境で比較をするのが妥当である。しかし、時間的都合などにより、従来法を実装するにはいたらなかった。そこで、本実験では、従来法の評価の際に計測されていた OPT_0 の実行速度を基準にし、それを 1 とみなしたときのそれぞれの実行速度の値を求めた。その値をグラフ化したものが表 35である。従来法ではこれにより、それぞれの最適化効果を測ることができ、それぞれのプログラムにおいて実行速度がどれほど改善されるかがわかる。

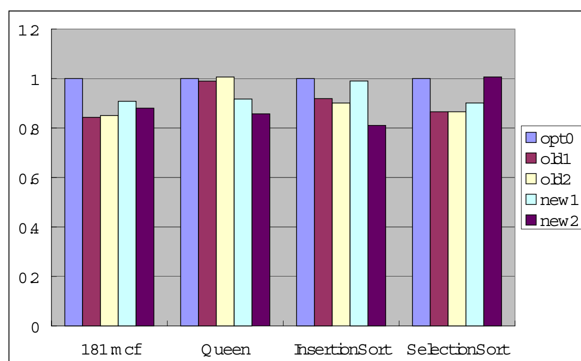

図 35: 本手法と従来法の実行時間の相対比

6.4　実験2の考察

従来法ではほとんど改善できなかった queen は、本手法では最適化効果を示すことのできた例で、変更文の移動が最適化効果を向上させることを示している。181.mcf のような大きなプログラムでは、ほぼ同等の効果がみられた。Insertion Sort や Selection Sort をでは、一概にどちらかが良いとは言い切れないが、NEW_1 と NEW_2 の結果からコピー伝播と本手法を繰り返し適用することで、本手法の最適化効果が大きくなることも期待できる。

6.5　考察と改善案

本手法を実装・評価した際に最適化効果を下げてしまう場合を発見した。本節では、それについての考察と改良案について述べる。

発見することの出来た最適化効果を妨げる場合は以下のようにまとめられる。

1. 部分冗長から全冗長にするために起こる計算量の増加
2. コード移動によるレジスタ圧力の増加

いずれの場合も変更文を移動することによって誘発される事例であり、変更文を移動させずに部分冗長性除去をした場合と比べて損をしている。1の問題点の改良案については、改善案を提案することが出来ている。2の問題は時間的問題もあり実装にいたっていない。今後の課題にする予定である。

6.5.1　計算回数の増加

変更文を移動すると計算回数が増えてしまう場合がある。図36(左) の $a_0 + b_0$ の計算は、ブロック 2 でのみ計算される計算式であった。分岐は 4 つに分かれているため、それぞれのパスを通る確率を 1/4 とすれば、最適化前の $a_0 + b_0$ の計算量は 1/4 となる。変更文を移動しない部分冗長性除去の場合、コード移動は変更文に妨げられ、計算式の挿入は 1 つである。そのため、この場合の $a_0 + b_0$ の計算量は 1/4 となり最適化前と同等である。

本手法では、図 37のように、変更文を移動することができるため、$a_0 + b_0$ の *Earliest* がブロック 1 へと引き上げられる。そのため、*Delay* の結果はブロック 1 の後続ブロック全てに対して *true* となってしまう。従って、全ての分岐文の後続ブロックで必ず $a_0 + b_0$ が計算されることとなるので計算量は 1 となり、最適化前よりも増加してしまう。

この問題を解決するたには、挿入された計算式が支配するブロックの命令、および、支配辺境における ϕ 関数の右辺の引数に何度使用されているかを調べればよい。使用がない場合はその挿入されたコードを除去すればよい。例えば、図37のブロック 3 で挿入された計算式が削除できるか見るためには、ブロック 3 が支配するブロックを求める。この場合であるとブロック 3 が支配するブロックはブロック 3 のみである。また、ブロック 3 を通るパスにおいて支配辺境となるブロックはブロック 6 である。そのため、t_1 が使用されている回数を数えるとブロック 3 で 0 回、支配辺境において、t_1 が引数となる ϕ 関数はないのでこの計算式を削除することが出来る。

6.5.2　コード移動先の計画性

変更文を移動する際に考慮しなければならない性質に *Dominance Property* がある。これは、SSA 形式の性質上、変数の定義はその使用を必ず支配する場所に置かなければならない、という意味である。この性質を守るような変更文の移動可能な場所はプログラムの実行パス上で実行される確率の高い場所となってしまう。

例えば、図37における b_0 は、最適化前はブロック 2 にあった定義文である。b_0 の *ModEarliest* となる場所は、支配木の親である分岐命令のあるブロック 1 となる。ここで、$a_0 + b_0$ の計算式のコード移動の結果が図37 の右のようになったとすれば、b_0 の *ModLatest* が *Dominance Property* を守る場所はブロック 1 になってしまう。6.5.1節では、無用な計算式の挿入をしてしまう場合を述べたが、無用な計算式を削除するタイミングを *ModLatest* を求める前で行うか、後で行うかで最適化効果が変わってくる。無用な計算式を削除した後で *ModLatest* を求めると、最適化効果は変更文を移動しない部分冗長性除去と同等になると考えられる。

7　まとめ

7.1　結論

本研究では、SSA 形式上の部分冗長性除去を制御フローグラフの上でのデータフロー方程式を順次求めることによって実現する見通しの良いアルゴリズムを提案した。また、部分冗長となる計算式の変更文を移動させることによって、コード移動が可能となる範囲を広げ、より最適な計算式の挿入ポイントを探す部分冗長性除去アルゴリズムを提案した。立川の方法 [15] は SSA 形式上で制御フローグラフを用いる第一歩であったが、一時変数に対する ϕ 関数の扱いが大変複雑であった。また、その方法は、変更文を移動しないことを前提としていたため、変更文移動による同一 ϕ 関数内の生存区間干渉に対応できなかっ

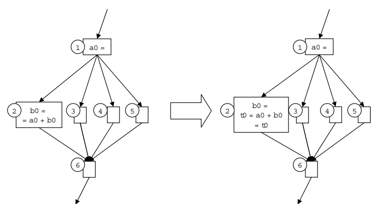

図 36: 従来の SSAPRE で最適化した例

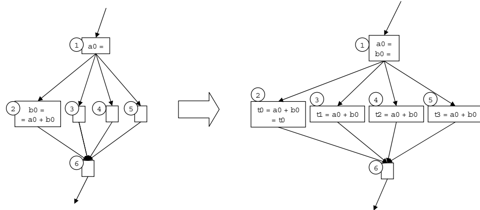

図 37: New PRE で最適化した例

た。しかし、本研究では、これらの問題を克服し、、同一 ϕ 関数内の生存区間干渉を誘発する他の SSA 最適化後でもこの部分冗長性除去を適用出来るようになった。

実験から得られた結果は、本アルゴリズムによる最適化は、ループ不変式のループ外移動と共通部分式除去を同時に適用した場合とほぼ同等もしくはそれ以上であり、効果があると言える。また、立川の方法 [15] ともほぼ同等の最適化効果を示せ、プログラムによっては最適化効果が大幅に改善できたものも見受けられる。

変更文を移動させることによる最適化の効果については、まだ検討の余地があるが、提案手法の該当部分をはずすことで、変更文を移動しないアルゴリズムも容易に実現できる見込である。このようなバリエーションを包括した一般的な枠組を提案したことは意義があるものと考える。

7.2 今後の課題

変更文の移動については、本アルゴリズムによる、ロード命令の数が増えることに対処するアルゴリズムが必要である。ロード命令が増えてしまう理由は、計算式の挿入による計算回数の増加と変更文の移動による生存変数の局所的な存在数の過多によるものであると考えられる。これらを解決することによってロード命令の数を減らすことに成功すればより実行時間に改善が見られるのではないかと考える。

5節で述べたように、今後は

1. 変更文移動のアルゴリズムの改善
2. 冗長効果の高い計算式の選択

などについて検討していきたい。

謝辞

本研究の一部は、文部科学省科学技術振興調整費、日本学術振興会科学研究費補助金、財団法人栢森情報科学振興財団の補助を受けた。

参考文献

[1] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 1–11, January 1988.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[3] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 159–170, 1994.

[4] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 98–105, 1982.

[5] Fred C. Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 273–286, 1997.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 451–490, October 1991.

[7] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 3, pp. 627–676, 1999.

[8] J. Knoop, O. Rüething, and B. Steffen. Lazy code motion. *SIGPLAN Notices*, Vol. 27, No. 7, pp. 224–234, June 1992.

[9] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 4, pp. 1117–1155, July 1994.

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

[11] 文部科学省. 科学技術振興調整費 総合研究「並列化コンパイラ向け共通インフラストラクチャの研究」. http://www.coins-project.org/.

[12] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 12–27. ACM Press, 1988.

[13] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vasta Santhanam. Translating out of static single assignment form. In *Proceedings of the 6th International Symposium on Static Analysis*, Vol. 1694 of *Lecture Notes in Computer Science*, pp. 194–210. Springer-Verlag, 1999.

[14] 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, November 1997.

[15] 立川英. 静的単一代入形式上の部分冗長性除去. Master's thesis, 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, 3 2004.