別名情報に基づくレジスタ促進

滝本宗宏[†] 原田賢一[††]

静的単一代入形式をはじめとするプログラム解析用のプログラム形式は，多くの有効かつ複雑な解析アルゴリズムの実現を可能にした．これらのプログラム形式のほとんどが変数の定義と使用の関係に基づいているので，解析の精度は，変数へのアクセスをどれほど詳細に明示化できるかに依存する．一般に，変数の定義と使用の明示化は，レジスタ促進によって仮想レジスタレベルで行われることが多い．仮想レジスタは，その定義あるいは使用が不明な場合に，対応する変数からのロード命令またはストア命令の挿入がそれぞれ必要になる．このとき，解析に利用可能な定義と使用は，ロード命令からストア命令までの範囲内に限定される．したがって，その範囲を拡大することは，解析の詳細化に直接貢献するので，冗長なロード/ストア命令を除去することが重要である．本研究は，従来考慮されなかった May 別名の 1 種である部分 Must 別名に基づいて，冗長なロード命令を除去する手法を提案する．従来，ロード命令の除去は，ロード対象になる記憶場所への参照式が同じものだけを取り扱ってきた．これに対して，部分 Must 別名に基づく冗長除去は，同じ記憶場所を表しているすべての別名を対象にすることができる．本手法では，また，部分 Must 別名を Must 別名に変換して，間接参照除去を同時に行うことができる．本手法の実現は，ビットベクタ表現を用いた単方向データフロー解析を用いて行うことができるので，プログラムサイズを n とした計算量は，$O(n^2)$ で抑えられる．

Register Promotion Based on Alias Information

Munehiro Takimoto[†] and Kenichi Harada[††]

Program forms e.g. the static single assignment form have brought about development of a variety of effective but complex algorithms for static analyses or code optimizations. Since these forms are based on relations between definition and use of variables, called def-use relation, their availability depends on how precisely accesses to memory locations can be represented explicitly. In general, explicit representation of def-uses are often realized by register promotion techniques, which allocate variables to virtual registers. In the virtual register form, accesses to unknown memory locations are explicitly represented by load and store instructions, which restrict the region for available def-use relations. Therefore, removing load and store which expand such region contributes to improvement of traditional program analyses on accuracy of their results. We propose a new technique to remove redundant load operations from virtual register forms based on partial must-aliases and show its effectiveness by experimental results. Our approach can detect the redundant loads of variables aliasing with a unique memory location, while in conventional techniques, detection of them is limited to same access pattern. Furthermore, our technique has effectiveness to transform a part of may-aliases into must-aliases. Since our approach can be implemented by simple data flow analysis using bit-vector representation similar with partial redundancy elimination, its complexity is given by $O(n^2)$ pessimistically.

1. はじめに

レジスタ割付け（register allocation）[1),3),19)]は，最適化コンパイラにおける重要な処理の 1 つである．レジスタ割付けを行うにあたって，まず，割付けの対象になる変数を決定する必要がある．コンパイラは，レジスタ割付けの候補を表現するために，無限個数の**仮想レジスタ**（virtual register）を用いることが多い．変数が仮想レジスタに割り付けられると，その変数の定義と使用は，それぞれ仮想レジスタへの値の転送，参照として表現されることになる．以降，これらのレジスタ操作を，変数の場合と同様に，それぞれ仮想レジスタの定義，使用と呼ぶ．仮想レジスタは，物理的なレジスタと同じ性質をもっていることを前提にして

† 東京理科大学理工学部情報科学科
Department of Information Sciences, Faculty of Science and Technology, Tokyo University of Science
†† 慶應義塾大学理工学部情報工学科
Department of Information and Computer Science, Faculty of Science and Technology, Keio University

```
1       int x;
2
3       func1 () {
4         x = x + 1;
5         func2 ();
6         printf ("%d",x);
7       }
```

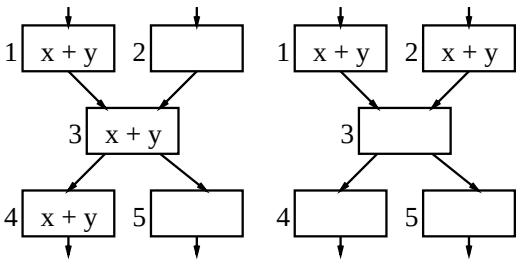
図 1 仮想レジスタとロード/ストア命令
Fig. 1 Virtual registers and loads/stores.

いるので，**別名**（alias）が用いられる場合に，正確な別名情報が求められていなければ，その別名への定義や別名による使用を，元の変数 v に割り付けられた仮想レジスタ r を用いて表現することができない．このため，v が別名によって定義される可能性があるプログラム点においては，その直後に v（メモリ）の値を r へロードしておく必要がある．また，v が別名によって使用される可能性があるプログラム点においては，その直前に r の値を v へストアしておく必要がある．このロードとストアは，両者に挟まれた範囲において別名による影響を受けないことを保証する．この性質は，定義–使用関係に基づくコード最適化の実現を容易にかつ効果的にするという利点をもつ．変数を仮想レジスタに割り付ける操作は**レジスタ促進**（register promotion，以降単に促進と呼ぶ）と呼ばれている．

例：図 1 の関数 `func1` において，変数 `x` を仮想レジスタ r に促進することを考える．`func1` の先頭においては，`x` の定義が分からないので，4 行目の文の直前に，x から r へのロード命令が必要である．5 行目の関数呼出しにおいては，`x` の値が他の関数によって使用される可能性があるので，その前に `r` から `x` へのストア命令が必要である．さらに，6 行目において，`x` が使用されているので，6 行目の関数呼出しの直前に，`x` から `r` へのロード命令が必要になる．

最終的にコード生成部が生成する機械コードが高速に実行されるようにするためには，レジスタ促進の際に挿入されるロード命令とストア命令の実行頻度を低くする必要がある．また，仮想レジスタに割り付けられた変数の定義–使用関係に基づいて行う最適化においては，変数はレジスタ参照によって表現されるので，ロード命令から対応するストア命令までの範囲が広いほど，高い効果が期待できる．各仮想レジスタに対するロード命令とストア命令の実行頻度を低くし，両者の間隔を広げる効果的な手法として，**部分冗長除去**（partial redundancy elimination，以降，PRE と呼ぶ）[8),14),15),18)]を用いた方法[17)]がある．

PRE は，あるプログラム点 n に存在する式が，n



(a) 冗長な式と部分冗長な式 redundant or partially redundant expressions
(b) 冗長除去の結果 Result of eliminating redundancy

図 2 部分冗長除去
Fig. 2 Partial redundancy elimination.

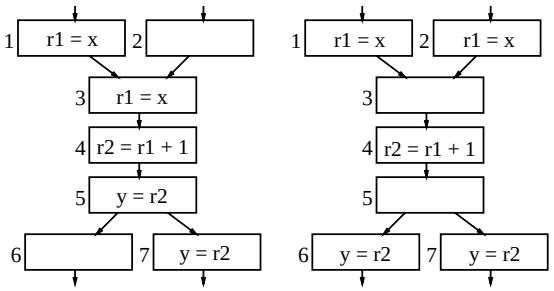
を通るすべての実行パス上で冗長（**全冗長**，totally redundant）である場合には，その式を除去し，一部の実行パス上で冗長（**部分冗長**，partially redundant）である場合には，冗長にならない実行パスだけが通る上方のプログラム点へ元の式を挿入することによって，n に存在していた式を全冗長なものにして除去する．この式の挿入と除去は，プログラムの上方へ移動していることに等しいので，それらをまとめて**巻上げ**（hoisting）と呼ぶ．

例：図 2 (a) に示すフローグラフにおいて，節 4 の式 `x+y` は，節 3 の式 `x+y` に対して全冗長なので，共通部分式としてプログラムの意味を変えずに除去することができる．これに対して，節 3 の式 `x+y` は，節 1 からの実行パス上でだけ冗長となる部分冗長である．部分冗長な式をそのまま除去すると，元の実行パス（図 2 (a) の節 2，3，5 を通る実行パス）では，計算していた式がなくなってしまうので，部分冗長な式は直接除去することができない．この例の場合には，節 2 に `x+y` を挿入することによって，節 3 の `x+y` は全冗長になり，除去可能になる．このように，PRE は部分冗長を全冗長に変えることによって，すべての全冗長性を除去する．図 2 (a) のプログラムは，PRE によって図 2 (b) のように変換できる．

PRE による冗長除去のアプローチは，適用対象を，ロード元の変数によって区別されるロード命令とすることによって，冗長なロード命令の除去にも適用することができる．

例：図 3 (a) における節 3 の変数 `x` から仮想レジスタ `r1` へのロード命令 `r1=x` は，節 1 のロード命令に対して部分冗長であり，図 3 (b) に示すように節 2 へ巻き上げることができる．結果として，節 1，3 を通る実行パス上のロード命令の数が 2 から 1 になる．

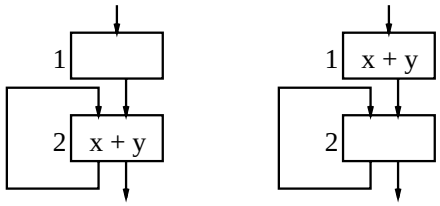
ストア命令に対する PRE は，辺の向きを逆にしたフローグラフへの適用によって実現できる．この場合，



(a) 部分冗長なロード/ストア命令
Partially redundant load /store

(b) PRE の結果
Result of PRE

図 3 ロード/ストア命令に対する PRE
Fig. 3 Applying PRE to load/store.



(a) ループ不変式
Loop-invariant expression

(b) ループ不変式のループ外移動
Moving loop-invariant expression out of loop

図 4 PRE によるループ不変式のループ外移動
Fig. 4 Moving loop-invariant expression out of loop by PRE.

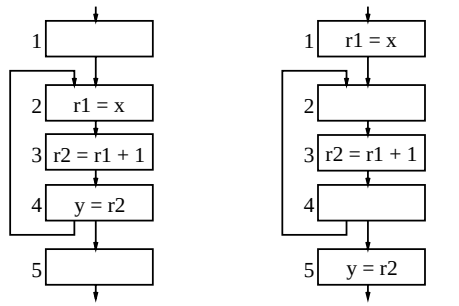
ストア命令は，後続の節へ移動することになる．巻上げと逆の操作にあたる，この移動を**降下**（sinking）と呼ぶ．

例： 図 3 (a) における節 5 の仮想レジスタ `r2` から変数 `y` へのストア命令 `y=r2` は，節 7 のストア命令に対して部分冗長であり，図 3 (b) に示すように節 6 へ降下させることができる．結果として，節 5，7 を通る実行パス上のストア命令の数が 2 から 1 になる．

ロード命令に対する PRE とストア命令に対する PRE は双対の関係にある．

また，PRE には，ループ内の不変式を部分冗長なものとしてループの外へ巻き上げる効果もあるので，式の実行頻度を大きく減らすことができる．

例： 図 4 (a) における節 2 の `x+y` は，節 1 から節 2 へ到達する実行パス上では冗長ではないが，ループの**戻り辺**（back edge）を通って節 2 へ到達する実行パスでは，節 2 の式自身との間で冗長であると見なすことができるので，部分冗長となる．したがって，節 2 の式は，巻上げによって図 4 (b) に示すように変形できる．



(a) ループ不変ロード/ストア命令
Loop-invariant load/store

(b) ループ不変ロード/ストア命令のループ外移動
Moving loop-invariant load/store out of loop

図 5 PRE によるループ不変ロード/ストア命令のループ外移動
Fig. 5 Moving load/sotre out of loop by PRE.

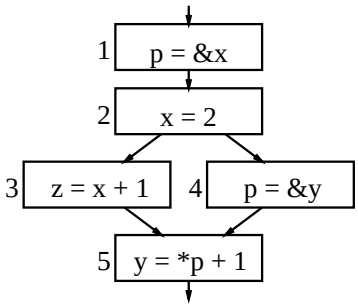


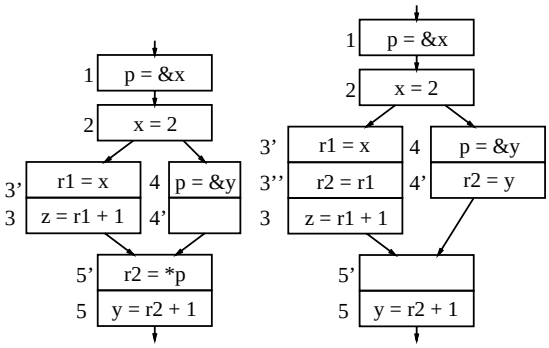
図 6 別名の例
Fig. 6 Example for aliases.

このループ外への移動による効果は，ロード命令に対する PRE でも同様であり，ループ不変ロード命令をループの上方に移動させることができる．一方，ループ不変ストア命令は，ロード命令と双対関係から，ループの下方へ移動させることができる．

例： 図 5 (a) における節 2 のロード命令は，PRE によって節 1 へ移動でき，ストア命令は，ロード命令との双対関係から，節 5 へ移動できる．

このように，ロード命令やストア命令への PRE の適用は，仮想レジスタ上で扱える定義–使用関係の範囲を広げ，ロード命令やストア命令の実行コストを下げることに有効に働く．しかし，PRE が，字面上で一致した命令を対象として冗長性を除去する手法であることから，同じメモリ領域を表す別名との間に生ずる冗長性は扱うことができないという問題があった．

例：図 6 において，`x` と `*p` に対して従来のレジスタ促進を適用し，それらのロード命令だけを示すと，**図 7** (a) に示す結果が得られる．これ以降も図 6 を例に用いるが，説明を容易にするために，`p` の促進は行われないものとする．図 7 の節 3 を通る実行パスにおいて `*p` と `x` とは別名であるので，節 5' の `*p` に対するロー



(a) 従来のロード命令の挿入 Naive insertion of load　(b) 部分 Must 別名に基づいたロード命令の除去 Naive insertion of load

図 7 別名を含むロード命令の冗長除去
Fig. 7 Removing redundant aliased load.

ド命令は，x に対するロード命令が節 3' にあることを考えると，部分冗長である．しかし，これらのロード命令は，字面上異なる変数に対するものなので，PRE ではその冗長性を取り除くことができない．

本研究では，同じメモリ領域を表す別名間で，冗長なロード命令を除去し，より広範囲なレジスタ促進を可能にする手法を提案する．ストア命令に関しては，従来法を適用するものとする．

あるプログラム点においてつねに同一のメモリ領域を表す別名を **Must 別名**（must aliases）と呼ぶ．これに対して，別名がどのメモリ領域を指すのかが一意に特定できない別名を **May 別名**（may aliases）と呼ぶ．May 別名の情報は，同一のメモリ領域を表す変数で置き換えることができないので，後のプログラム解析やコード最適化に対しては，保守的な取扱いを厳守する立場から，それらの効果を抑制する働きしかしない．

本手法では，この May 別名を次の 2 種類に分類し，性質の良いものを積極的に利用する．

絶対 May 別名（absolute may-aliases）： プログラム点にどのような値が到達するのかを静的に知ることができないので，変数が表すメモリ領域を特定できない別名

部分 Must 別名（partial must-aliases）[24]**：** プログラム点に到達できる実行パスが複数存在する場合でも，その中の少なくとも 1 つの実行パスについては変数が表すメモリ領域を特定できる別名

絶対 May 別名は，冗長性の除去によって，どれほどの変数が影響を受けるかを知ることができないので，本手法の対象とはしない．一方，部分–Must 別名は，Must 別名となるプログラム点へ巻き上げることによって，冗長性の除去を行う候補とすることができる．

例： 図 7 (a) において，節 5' に現れる `*p` は，節 3 を通る実行パス上では，変数 `x` への参照を意味し，節 4 を通る実行パス上では，変数 `y` への参照を意味するので，部分 Must 別名である．この `*p` は，`*p` を含むロード命令 `r2=*p` を節 3'' と節 4' へ巻き上げることによって（図 7 (b)），それぞれ `x`，`y` の Must 別名へ変換することができる．結果として，ロード命令 `r1=x` と `r2=*p` との間でロード元が全冗長となり，`r2=*p` をレジスタ転送命令 r2=r1 で置き換えることによって，ロード命令の冗長性を除去できる．

上の例のように，部分 Must 別名に対するロード命令を巻き上げた後，Must 別名に対するロード命令間の冗長性を除去することによって，Must 別名だけでなく，部分 Must 別名が用いられている場合でも，ロード命令の冗長除去を実現することができる．さらに，本手法には，巻上げにともなうロード命令の挿入の際に，ロード対象になる別名の中から，最も単純な参照式を選ぶことによって，ロードに要するコストを下げる効果ももつ．

例： 図 7 (b) において，節 4' に巻き上げられたロード命令 `r2=*p` は，ロード元の `*p` が `y` の Must 別名であることから，`*p` の代わりに，よりコストの低い `r2=y` で置き換えることができる．

本手法は，次の 2 段階で実現する．

可能な巻上げ解析： 巻き上げることが可能な最大範囲を計算する．

可能な巻戻し解析： 不要に巻き上げたロード命令を巻き戻すことが可能な範囲を計算する．

本稿における以降の構成は，次のとおりである．2 章で，前提とするプログラム表現と解析について簡単に述べる．そして，3 章において，Must 別名と部分 Must 別名の関係について述べる．次に，4 章において，間接参照の巻上げと不要な巻上げの巻戻しについて述べ，5 章で，プログラム変換によって実際にどのように冗長性を除去するのかを述べる．6 章において，評価実験による本手法の効果を示すとともに，本手法全体の計算量について考察する．7 章で関連研究を示し，最後に結論を述べる．

2. 入力プログラム

本手法の適用に際しては，原始プログラムに対応する**制御フローグラフ**（control flow graph，以降 CFG と呼ぶ）がすでに作成されているものとする．CFG は，単一の文からなる節の集合 N，辺の集合 $E \subset N \times N$，特別な節である開始節 **s** と終了節 **e** からなる四つ組

$(N, E, \mathbf{s}, \mathbf{e})$ である．CFG 節 n について，その先行節を $pred(n)$ で表し，後続節を $succ(n)$ で表す．

プログラム中に現れるデータ参照，すなわち変数の定義および参照は，対応するメモリ領域へのアクセスを表現するために，次の BNF で規定される $indir$ の構文の式によって表現する．この式を以降，**参照式**と呼ぶ．$indir$ は，参考文献 21) において採用さているデータ参照のパターンを拡張したものである．

$base \rightarrow$ `pointer_constant`
　　| `address_of_variable`

$addr \rightarrow base$
　　| $base$+`const`
　　| $base$+`const`×`var`

$indir \rightarrow * addr$
　　| $** base$

C 言語における，基本データ型の変数 `x` への参照は，`x` に対応するメモリ領域のアドレスを `x_addr` とするとき，`*x_addr` で表現する．また構造体のフィールドへの参照（たとえば `a.x`）や配列要素への参照（たとえば `a[i]`）は，それぞれ参照式 $*(base+\texttt{const})$，$*(base+\texttt{const}\times\texttt{var})$ の形で表現できるものに限定する．ここで，参照式の左辺値の表現には，$addr$ によって示される記法を用いる．これを**アドレス式**と呼ぶ．以下の図や説明においては，参照式の表現に通常の C 言語の記法を用いる．

CFG において，節 i 中に出現する参照式 e の仮想レジスタ r への促進に関して，次の変換がすでになされているものとする．

e の使用（変数参照）の場合： i の直前にロード命令 r=e を挿入し，i 中の e を r で置き換える．

e の定義（代入先）の場合： i の直後にストア命令 e=r を挿入する．

本手法は**手続き内**（inter-procedural）の解析を行うので，関数呼出しが行われる場合には，変数（メモリ）と仮想レジスタとの間で，値の一致性を保証するために次の命令の挿入を仮定する．すなわち，参照式を e_g，対応する仮想レジスタを r_g とするとき，e_g が次の条件を満たす場合，関数呼出しの前にストア命令 e_g=r_g が挿入され，関数呼出し後にロード命令 r_g=e_g を挿入されているものとする．

(1) e_g が大域変数への参照を表す．
(2) 関数呼出しにおいて局所変数のアドレスを渡す実引数が用いられ，e_g がその局所変数に対する参照式である．

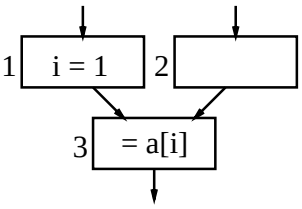


図 8 定数畳込みによる Must 別名
Fig. 8 Must-alias based on constant folding.

ここで挿入したロード命令とストア命令は，冗長性の除去を行う際の初期状態に含まれる．

例： 図 6 に対するロード命令の初期状態は，図 7 (a) に示すとおりである．

本手法を適用する以前に，別名解析は終了していて，別名情報は利用できるものとする．別名解析は，一般に制御フローを考慮しない**フロー非依存別名解析**（flow-insensitive alias analysis）と，制御フローを考慮に入れた**フロー依存別名解析**（flow-sensitive alias analysis）[19]とに大別できる．本手法では，フロー依存別名解析が行われていることを仮定する．

本手法はコード巻上げに基づいているので，**クリティカル辺**（critical edge）[14),15)]合成節の挿入によって取り除かれているものとする．

また，2 つ以上の先行節をもつ節に入ってくる辺は，すべて空文にあたる節が挿入されているものとする．この変形によって，クリティカル辺の問題は解決でき，後のデータフロー解析を単純にすることができる☆[14)]．

3. Must 別名と部分 Must 別名

本手法は，冗長性を除去するロード命令の候補を，Must 別名と部分 Must 別名をもつ参照式にまで拡張するものである．この節では，Must 別名の情報について説明し，Must 別名と部分 Must 別名の関係を述べる．

3.1 Must 別名

本手法では，Must 別名の関係にある変数についても，それらのロード命令間の冗長性を考慮するので，本手法の有効性は Must 別名解析の精度によって左右される．そのため，解析の前にすでに**定数伝播**（constant propagation）が行われ，参照式における定数オペランドをもつ部分式は，すべて畳込みが行われているものとする[1)]．

例： **図 8** の `a[i]` に対する別名解析の結果は，節 1 の

☆ 説明を簡単にするために，本稿の図や例では，必要な空節だけを表示する（図 7 の節 4'）．

node	$MUST$
1	$true$
2	$true$
3	$true$
3’	$true$
4	$true$
4’	$true$
5	$false$
5’	$false$

図 9 図 7 (a) の Must 別名情報
Fig. 9 Must-alias information for Fig. 6.

`i=1` によって，節 1 において `a[1]` と `a[i]` が Must 別名になる．`a[i]` に対する参照式を $*(\mathtt{a}+4\times\mathtt{i})$ とすると，節 1 において `i=1` によって畳み込まれる $*(\mathtt{a}+4)$ を $*(\mathtt{a}+4\times\mathtt{i})$ の Must 別名として扱う．

以降，$MUST_{(x,n)}$ は，別名解析の結果に基づいて，プログラム点 n における参照式 x が Must 別名をもつ場合に $true$，そうでなければ，$false$ となる述語であるとする．

例： 図 7 (a) の各節 n における `*p` の $MUST_{(*p,n)}$ は，**図 9** に示すとおりである．

3.2 部分 Must 別名

参照式 x がプログラム点 n において部分 Must 別名であるとは，開始節 **s** から n に到達する実行パス上で，x が Must 別名になるプログラム点 n' が少なくとも 1 つ存在することを意味する．また，部分 Must 別名となる参照式 x は，その部分式として必ず変数 v の使用む．したがって，**s** から n' への実行パス上には，v の定義が存在し，その定義点から n' までのすべての実行パス上で x は Must 別名となる．v の定義点よりもさらに上方への参照式の巻上げは，プログラムの意味を変える可能性がので，そのような巻上げは禁止する．ここでは，部分 Must 別名である x を，巻上げが可能なプログラム点のうちで，最も開始節に近い点へ巻き上げ，x を Must 別名に変換することにする．

4. 巻上げ解析

本手法は，PRE のデータフロー解析を別名関係に応用することによって実現できる．本稿では，PRE の 1 手法である**遅延コード移動**（lazy code motion，以降 LCM と呼ぶ）[14] に拡張を加えることによって実現する．LCM は，次の 2 段階のデータフロー解析で実現される．

(1) 可能な巻上げ点の解析

(2) 不要な巻上げを巻き戻す点の解析

まず，(1) の解析によって，巻上げが可能なプログラム点を計算し，そのうちで開始節に最も近いプログラム点を求める．しかし，これらのプログラム点の中には，不要なものが含まれているので，不要な巻上げを行わないように (2) の解析で巻戻しを行う．不要な巻上げは，変数の生存期間を無駄に長くする可能性があり，後のレジスタ割付けにおいて，レジスタからのスピルを引き起こす危険性がある [6]．

本手法も，可能な巻上げ点の解析と巻戻し点の解析によって実現する．3.2 節で述べたように，部分 Must 別名は，巻上げが可能なプログラム点のうちで開始節に最も近い点に巻き上げることによって，Must 別名への変換による効果をすべて得ることができるので，可能な巻上げ点の解析には LCM と一致したものを用いる．一方，巻戻し点の解析には，異なる方法を用いる．実際の巻上げ点は，巻戻しを行ったあとに得られるプログラム点によって決定されるので，巻戻しは，Must 別名への変換による効果を低減させない範囲で行わなければならない．

本章では，可能な巻上げ点の解析を説明したのち，Must 別名への変換効果をもつ可能な巻戻し点の解析について述べる．

4.1 可能な巻上げの解析

参照式 x についての可能な巻上げ点の解析は，次に示す方程式 1 の最大解を求めることによって，得ることができる．

方程式 1（可能な巻上げ）

$$HOISTED_{(x,n)} =_{def} \begin{cases} false & if\ n\ is\ \mathbf{e} \\ COMP_{(x,n)} \vee TRANSP_{(x,n)} \wedge \prod_{m\in succ(n)} HOISTED_{(x,m)} & otherwise \end{cases}$$

述語 $HOISTED_{(x,n)}$ は，参照式を x とするロード命令がプログラム点 n まで巻き上げることができることを表す．述語 $COMP_{(x,n)}$ は，解析対象になる CFG において，参照式を x とするロード命令がプログラム点 n に出現していることを表す述語である．$TRANSP_{(x,n)}$ は，参照式 x やその部分式がプログラム点 n で変更されないことに加え，x が表すメモリ領域中の値も変更されないことを表す述語である．

$HOISTED$ は，終了節 **e** については $false$，それ以外のすべての節については $true$ に初期化し，後向きに伝播させる．$TRANSP = false$ である節においては，参照式が表すメモリ領域やそのメモリ領域中

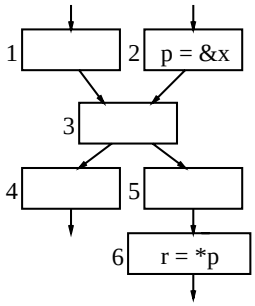


図 10 安全な巻上げ
Fig. 10 Safe hoisting.

node	$HOISTED$
1	$false$
2	$false$
3	$true$
3'	$true$
4	$false$
4'	$true$
5	$false$
5'	$true$

(a) 可能な巻上げの解析結果
Result of possible hoisting

node	$PREINSERT$
1	$false$
2	$false$
3	$false$
3'	$true$
4	$false$
4'	$true$
5	$false$
5'	$false$

(b) 仮の挿入点
Pre-insertion points

図 11 可能な巻上げの解析結果と挿入点
Fig. 11 Result of possible hoisting and insertion.

の値が変わる可能性があるので，伝播をブロックしなければならない．また，巻上げは，各実行パスに新しいロード命令を挿入しない範囲で行う必要がある．この性質は，一般に**安全**（safty）性と呼ばれ，対象とする計算がすべての隣接節（前向き移動の際は，すべての先行節，後向き移動の際は，すべての後続節）から，移動してくる場合に限って，さらに隣接節への移動を許すという条件で表現される．方程式 1 において，プロダクション $\prod$ は，この安全性の条件を表している．
例： 図 10 の節 6 の `r=*p` を，節 1 や節 2 に巻き上げると，節 3 と節 4 を通る実行パスに新しい計算を導入することになる．したがって，図 10 の例の場合，このロード命令を安全に巻き上げることができる範囲は，節 5 と節 6 である．

最終的に，ロード命令は，その出現場所（$COMP = true$）から安全な範囲にあるプログラム点へ巻上げが可能となる．
例: 図 7 (a) の `r2=*p` に対する $HOISTED$ は，**図 11** (a) に示すとおりである．

上述の解析によって得られた，$HOISTED = true$ であるプログラム点のうち，開始節あるいは巻上げが $TRANSP = false$ によってブロックされた節に最も近いプログラム点をロード命令の挿入点として得ることができる．この挿入点を，最終的に得られる挿入点と区別して**仮の挿入点**（pre-insertion points）と呼ぶことにする．参照式 x に対するロード命令の仮の挿入点 n は，開始節あるいはブロックされた節から $HOISTED = false$ の節を前向きにたどっていって，最初に到達した $HOISTED = true$ の節として定義できる．開始節あるいはブロック点から始まる $HOISTED = false$ の節集合は，方程式 2 に示す $EARLIEST_{(x,n)}$ の最小解として求めることができる．

方程式 2（最も開始節に近い点）

$$EARLIEST_{(x,n)} =_{def}$$
$$\begin{cases} true & if\ n\ is\ \mathbf{s} \\ \sum_{m \in pred(n)} (\neg TRANSP_{(x,m)} \vee \neg HOISTED_{(x,m)} \\ \wedge EARLIEST_{(x,m)}) & otherwise \end{cases}$$

$HOISTED$ と $EARLIEST$ を用いて，$PREINSERT$ は方程式 3 として表現される．

方程式 3（仮の挿入点）

$$PREINSERT_{(x,n)} =_{def} HOISTED_{(x,n)} \wedge EARLIEST_{(x,n)}$$

例： 図 6 (a) の `r2=*p` に対する $PREINSERT$ は，図 11 (b) に示すとおりである．

4.2 可能な巻戻しの解析

仮の挿入点への巻上げによって，次の 3 つの効果が期待できる．

(1) 従来の冗長なロード命令の除去
(2) 部分 Must 別名から Must 別名への変換
(3) Must 別名関係にある冗長なロード命令の除去

ただし，不要なロード命令の巻上げは，本章のはじめに述べた理由から元の出現場所に向けて巻き戻しておく必要がある．その際，巻上げによって得られる効果を低減させないようにしなければならない．

従来の部分冗長除去における巻戻しでは，字面上で一致する式に出会うまで巻戻しをしていた．この方法を直接用いると，部分 Must 別名から Must 別名に変換された参照式が，再び部分 Must 別名になるプログラム点まで巻き戻される結果になり，(2) と (3) の効果を低減させる可能性がある．そこで，いったん Must 別名となった参照式のロード命令は，Must 別名である範囲内でだけで巻き戻すという条件を付加する．

巻戻しが可能な節の集合は，方程式 4 の最大解を求めることによって得ることができる．

node	$DELAY$
1	$false$
2	$false$
3	$true$
3’	$true$
4	$false$
4’	$true$
5	$false$
5’	$false$

(a) 可能な巻戻し解析の結果
Result of possible delaying

node	$INSERT$
1	$false$
2	$false$
3	$true$
3’	$false$
4	$false$
4’	$true$
5	$false$
5’	$false$

(b) 挿入点
Final insertion points

図 12 可能な巻戻し解析の結果と挿入点
Fig. 12 Result of possible delaying and final insertion.

方程式 4（可能な巻戻し）

$$DELAY_{(x,n)} =_{def}$$
$$\begin{cases} false & if\ n\ is\ \mathbf{s} \\ PREINSERT_{(x,n)} \vee \prod_{m \in pred(n)} ((MUST_{(x,n)} \vee \neg MUST_{(x,n)} \wedge \neg MUST_{(x,m)}) \wedge \neg COMP_{(x,m)} \wedge DELAY_{(x,m)}) & otherwise \end{cases}$$

述語 $DELAY_{(x,n)}$ は，参照式を x とするロード命令がプログラム点 n へ巻戻し可能であることを表す．$DELAY$ は，開始節 **s** については $false$，それ以外のすべての節については $true$ に初期化し，前向きに伝播させる．$DELAY$ 情報の伝播は，安全な移動を保証しなければならないので，先行節における $DELAY$ のプロダクション $\prod$ で表現する．ロード命令の節 n への巻戻しは，対応する参照式が Must 別名の場合，部分 Must 別名に戻らない範囲に限られる．この条件は，先行節を m として，$MUST_{(x,n)} \vee \neg MUST_{(x,n)} \wedge \neg MUST_{(x,m)}$ で表現できる．また，ロード命令は元の出現場所よりも先に巻き戻すことができないので，$DELAY$ の情報は $\neg COMP = true$ の場合にだけ後続節に伝播させる．
例： 図 7 (a) の `r2=*p` に対する $DELAY$ は，**図 12** (a) に示すとおりである．

最後に，可能な巻戻し解析の結果から，終了節に最も近いプログラム点が，実際にロード命令を挿入する箇所の候補として得られる．すなわち，参照式 x をもつロード命令の挿入点 n は，方程式 5 を満たす $INSERT_{(x,n)}$ によって与えられる．

方程式 5（挿入点）

$$INSERT_{(x,n)} =_{def} DELAY_{(x,n)} \wedge \neg \prod_{m \in succ(n)} DELAY_{(x,m)}$$

例： 図 7 (a) の `r2=*p` に対する $INSERT$ は，図 12 (b) に示すとおりである．

5. 冗長性の除去

前章で得られた解析の結果を用いてロード命令の除去と挿入を行う．まず，元のプログラムに存在しているロード命令をすべて除去する．次に，ロード命令を実際に挿入するプログラム点 n を決定するために，各プログラム点を通るすべての実行パス上にロード命令が先行して存在するという条件のもとで，$INSERT_{(x,n)} = true$ である節 n を求める．この性質は，**利用可能**（availability）性と呼ばれる．x を参照式とするロード命令が節 n で利用可能であることを述語 $AVAIL_{(x,n)}$ によって表す．$AVAIL$ は，前節で計算した情報だけを用いて方程式 6 のように定義できる．

$$AVAIL_{(x,n)} =_{def} \neg EARLIEST_{(x,n)} \wedge \neg DELAY_{(x,n)} \vee INSERT_{(x,n)}$$

$EARLIEST$ は，ロード命令を利用できないことが保証される開始節と $TRANSP = false$ のプログラム点から前向きにたどっていって，ロード命令に出会うまでの範囲を意味するので，利用可能性は，$EARLIEST$ の補集合に仮の挿入点を加えた，$\neg EARLIEST \vee PREINSERT$ で表現できる．さらに，この集合に対する可能な巻戻し点の補集合に挿入点を加えた，$(\neg EARLIEST \vee PREINSERT) \wedge \neg DELAY \vee INSERT$ が利用可能点の集合となる．$PREINSERT$ は，$INSERT$ に含まれるので，不要になる．

参照式を x，対応する仮想レジスタを r とするとき，そのロード命令の挿入点は，$INSERT_{(x,n)} = true$ である節 n であり，そのロード元の参照式は Must 別名情報と利用可能情報に基づく次の条件によって決定される．

(1) $MUST_{(x,n)} = true$：節 n における x の Must 別名である参照式の集合を $must(x,n)$ として，
  (a) $\{x' | x' \in must(x,n) \wedge AVAIL_{(x',n)}\} \neq \emptyset$ の場合：
  $x' \in must(x,n)$ に対応する仮想レジスタを r' として，レジスタ転送命令 r=r' を節の入口に挿入する．

表 1 PRE と本手法の適用結果
Table 1 Results of PRE and our approach.

program	comp	hoist-PRE	hoist-New	insert-PRE	insert-New	avail	must-PRE	must-New
sed	921	369	384	620	607	15	574	561
gprof	1036	415	429	654	656	6	593	602
gzip	2796	1221	1243	1804	1818	8	1717	1742
byacc	3616	1660	1682	2243	2256	7	2167	2193
make	2494	944	1012	1722	1734	29	1518	1557
ng	10630	4070	4158	7647	7682	37	7230	7296
lcc	7987	2272	2421	6719	6846	16	6226	6460

(b) $\{x'|x' \in must(x,n) \wedge AVAIL_{(x',n)}\} = \emptyset$ の場合：
$must(x,n)$ において計算コストが最も小さい参照式を x' として，r=x'を節の入口に挿入する．

(2) $MUST_{(x,n)} = false$：r=x を節の入口に挿入する．

例： 図 7 (a) において，`*p` に対する $INSERT$ は，節 3 と節 4 において $true$ となる．節 3 における `*p` の Must 別名が `x` であり，かつ $AVAIL_{(\mathtt{x},3)} = true$ であるので，節 3 に挿入する命令は `r2 = r1` となる．また，節 4 における `*p` の Must 別名が `y` であり，かつ $AVAIL_{(\mathtt{x},4)} = false$ であるので，節 4 に挿入する命令は `r2=y` となる☆．最終的に，図 7 (a) のプログラムは，参照式 `x` に対するロード命令も含めて，図 7 (b) に示す結果となる．

6. 評　　価

この章では，本手法の効果と解析コストについての評価実験の結果を示し，考察を与える．

6.1 本手法の効果

本手法の効果を示すために，従来の PRE を用いた手法と本手法を最適化部にもつ C コンパイラをそれぞれ実装し，評価を行った．これらのコンパイラのフロントエンドには，既成のコンパイラ LCC [12] を改良したものを用いた．フロントエンドは，目的コードに依存しない低水準の中間表現を生成し，ファイルとしてバックエンドに渡す．バックエンドでは，CFG を作成し，別名解析を行う．この CFG に対して，PRE と本手法を適用した．

評価には，7 つの有名なツール，sed，gprof，gzip，byacc，make，ng，lcc を使用した．**表 1** に評価結果を示す．表の各列の意味は次のとおりである．

comp： ロード命令の出現回数

☆ 図の簡素化のために，空節 4' の入口に対する `r2=y` の挿入は，節 4' への挿入によって表現している．

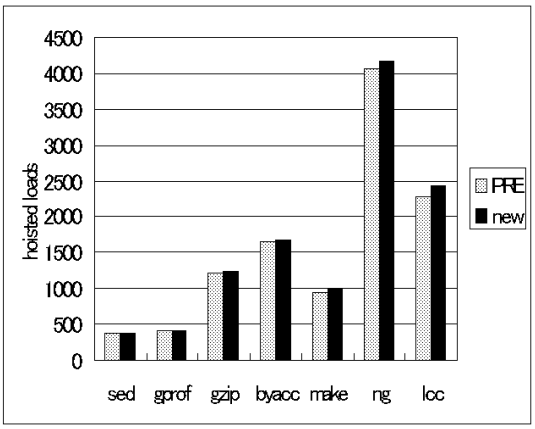


図 13 巻き上げられたロード命令数
Fig. 13 Number of hoisted loads.

hoist-PRE： PRE によって，巻き上げられたロード命令の個数

hoist-New： 本手法によって，巻き上げられたロード命令の個数

insert-PRE： PRE によって挿入されたロード命令の個数

insert-New： 本手法によって挿入されたロード命令の個数

avail： $INSERT_{(x,n)} = true$ の x が n において利用可能であることから，挿入が行われなかったロード命令の個数

must-PRE： PRE を適用した後に Must 別名である参照式に対応するロード命令数

must-New： 本手法を適用した後に Must 別名である参照式に対応するロード命令数

比較するために，プログラム中に出現したロード命令のうち，巻き上げられた命令の個数（hoist-PRE, hoist-New）をグラフにして**図 13** に示す．このグラフが示すように，すべてのプログラムについて，本手法の方が PRE を上回り，巻き上げられる元のロード命令の個数は，1%前後の改善が見られた．

次に，データフロー解析の結果，$INSERT_{(x,n)} =$

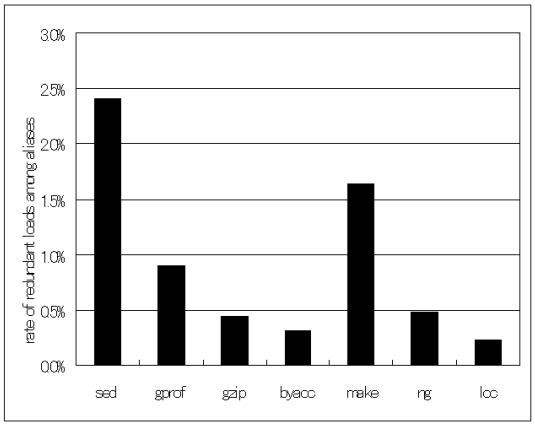


図 14 ロード対象の変数が別名関係にあることで，冗長になるロード命令の割合
Fig. 14 Rate of redundant loads with alias.

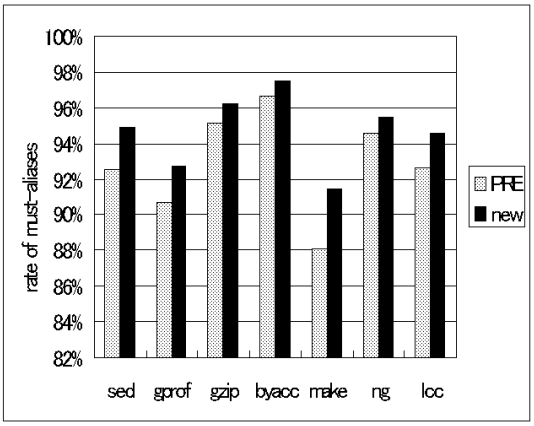


図 15 挿入するロード命令における Must 別名の割合
Fig. 15 Rate of insertions with must-aliases.

$true$ になった参照式 x についてプログラム点 n で利用可能な Must 別名が存在する割合 avail/(insert-New+avail)☆をグラフにして**図 14** に示す．このグラフが示すように，すべてのプログラムについて，$INSERT = true$ のうちで，さらに巻き上げられるロード命令のあることが分かった．本手法は，PRE と比べ，元から存在するロード命令の巻上げ数が増加するだけでなく，PRE によって挿入が行われるプログラム点よりもさらに巻上げを行う効果をもつ．

最後に，すべてのロード命令の挿入について，ロード対象になる変数が Must 別名である割合 must-PRE/insert-PRE，must-New/insert-New をグラフにして**図 15** に示す．このグラフが示すように，PRE に比べ，本手法の Must 別名の割合が大きくなっている．これは，本手法の方が，より低い実行コストのロード命令を生成する可能性が大きいことを示している．

6.2 解析コスト

本手法と PRE の計算コストを比較するために，両者が行うデータフロー解析における CFG 節の訪問回数を調べた．本実験で用いたデータフロー解析は，各参照式を 1 ビットで表現し，計算は 1 ワードごとにまとめて行う方法[13]で実現した．1 つの CFG 節に対応するビットベクタは，その長さがワード長を超える場合，複数のワードによって表現される．ここで，節 n の i 番目のワードを $w_{(n,i)}$ で表すことにする．計算に

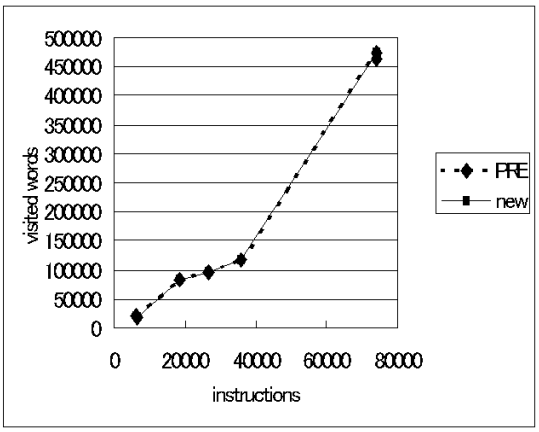


図 16 データフロー解析のコスト
Fig. 16 Cost for dataflow analyses.

よって $w_{(n,i)}$ が求められたとすると，計算前の結果と $w_{(n,i)}$ を比較し，ビットに変化が生じた場合に限って，節 n の先行節あるいは後続節 m のワード $w_{(m,i)}$ を次の計算候補としてワークリストに加える．このデータフロー解析の実現法は，ビットベクタの使用による並列計算の利点と，スロットごとの計算によって伝播の範囲を限定できる利点[10]を兼ね備えた手法として知られている[13]．

本実現において，CFG 節の訪問回数と各ワードへのアクセス回数の合計とは一致するので，実際の評価には，ワークリストに加えられたワード数を用いた．この結果をグラフにして**図 16** に示す．横軸は，中間表現における命令数を表し，グラフ上の各点は，表 1 に示したプログラムと同じ順序で対応している．このグラフが示すように，PRE と本手法とはアクセスするワードの数がほぼ一致する．これは，PRE と本手法

☆ insert-New は，実際に挿入されたロード命令の個数であり，$INSERT_{(x,n)} = true$ であっても，節 n で x の Must 別名が利用可能である場合には，そのロード命令の個数（avail）を含んでいない．したがって，$INSERT_{(x,n)} = true$ である挿入点の個数は，insert-New+avail となる．

が同等の計算コストで実現できることを示している．

プログラムサイズと解析コストの関係は，ほぼ線形に推移していることが分かる．PRE と本手法は，単方向のデータフロー解析によって実現できるので，プログラムサイズを n としたときに，計算量 $O(n^2)$ で抑えられることが知られている．実験結果から，本手法で行う解析はデータフロー解析の中でも高速で実践的なものといえる．

7. 関連研究

本手法のレジスタ促進は冗長性の除去に基づいている．式の冗長な出現を除去するために，従来から広く用いられてきた手法に**共通部分式**（common subexpression）の除去がある[1),3),19)]．これは，ある式 e のプログラム点 n における出現に関して，n を通るすべての実行パス上で，e が利用可能である全冗長な場合に，n での e を除去する手法である．共通部分式の除去では，部分冗長な式を除去することはできない．全冗長な式と部分冗長な式を除去し，さらにループ不変コードをループの外へ移動する手法として PRE がある．PRE は，Morel らがはじめて提案し[18)]，その後多くの変形手法が提案されてきた[8),15),16)]．PRE における解析は，前向きと後向きの依存をもつ双方向データフロー解析であるので，計算量は，プログラムサイズを n とすると，$O(n^3)$ であった．これに対して，PRE の解析を複数の単方向データフロー解析で得られる解の組合せで表現することによって，$O(n^2)$ の計算量を実現する手法が提案された[9),14)]．これらの中で，本手法は，単純な役割をもつ単方向データフロー方程式を組み合わせた LCM[14)] の手法に基づいている．

PRE は，プログラム中に現れる式の字面表現が一致するものに限って冗長除去の対象にするので，同じ値を生成する式でも，オペランド表現が異なっている場合には，冗長性を除去することができない．しかし，部分冗長除去を行った後，コピー代入を**コピー伝播**（copy propagation）[1)]によって除去すると，新たに字面が一致した式を生成できる．この効果は，PRE の**副次的効果**（second order effects）と呼ばれている．したがって，部分冗長除去とコピー代入を反復適用することによって，部分冗長除去の効果を高めることができる．この PRE とコピー伝播の反復適用の効果を効率的に得る方法として，式の木表現をフローさせる手法[22)]や多くの SSA 形式に基づく手法が提案されている[2),5),7),11),20)]．本手法における解析も滝本，原田が提案したデータフローグラフ[23),25),26)]を用いることによって，同等に高速化できるが，これらの手法は，各式ごとに解析を行うスロットワイズ手法[10)]に基づいているので，ワードサイズ分のビットに相当するデータを並列に処理できる本手法と比べて，どちらがコストが低いかは，解析対象となるプログラムに依存する．

部分冗長除去をレジスタ促進に応用した例として，Lo らの手法がある[17)]．Lo らは，実行時情報を利用して，安全でない巻上げでもループの外に出すことによって実行コストの低減が期待できる場合に，巻上げを行うという方針に基づいている．本手法では，安全でない巻上げによって，ロード命令をループの外へ移動させることはないが，制御フロー構造を変形することによって，安全に移動させることが可能になる[4)]．

本手法が前提にしている別名解析は，従来から多くの研究がなされてきた[1),3),19)]．別名解析は，データフロー解析を用いて，各プログラム点における別名情報を集めるフロー依存の方法がよく知られている．別名解析をどれだけ詳細に行う必要があるかは，後の最適化の効果にかかわる問題である．別名解析の精度を上げる方法の多くは，ポインタ変数がもつアドレス値の追跡を詳細に行うものである．これに対して，May 別名の 1 つである部分 Must 別名を Must 別名に変換するという異なった方法で別名情報の精度を上げるものに，滝本と原田が提案した **May 別名除去**（may-alias elimination）[24)]がある．部分 Must 別名と呼ぶ別名の性質は，この手法の中で初めて導入されたものである．

本手法は，部分 Must 別名に基づいて，PRE を別名間における冗長性の除去に拡張し，さらに May 別名除去の効果を合わせもつ手法といえる．

8. 結　論

本稿では，レジスタ促進において導入される仮想レジスタへのロード命令の冗長性を除去する手法を提案した．

本手法は，部分冗長であるロード命令に加え，Must 別名である変数からのロードに関して，その参照式の相違にかかわらず冗長性を除去することができることを示した．また，Must 別名でない変数に関しても，巻上げによって Must 別名へ変換できる部分 Must 別名を導入し，本手法によって部分 Must 別名が Must 別名に変換される効果があることを示した．この効果は，さらに多くの Must 別名間の冗長性を除去するだけでなく，挿入するロード命令の実行コストの低減にも通じるので，プログラムの実行速度向上に貢献する．

本手法は，LCM への簡単な変更によって実現でき，実質的な計算コストは PRE とほぼ一致することを示した．本稿の実験結果から，本手法は，従来の部分冗

長除去を用いた冗長なロード命令の除去と比べ，より強力な手法であるといえる．

参 考 文 献

1) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison Wesley (1986).
2) Alpern, B., Wegman, M.N. and Zadeck, F.K.: Detecting equality of variables in programs, *Proc. Principles of Programming Languages* (*POPL'88*), pp.1–11, ACM (1988).
3) Appel, A.W.: *Modern Compiler Implementation in ML*, Cambridge University Press (1998).
4) Bodik, R., Gupta, R. and Soffa, M.L.: Complete Removal of Redundant Expressions, *Proc. Programming Language Design and Implementation* (*PLDI'98*), pp.1–14, ACM (1998).
5) Briggs, P. and Cooper, K.D.: Effective Partial Redundancy Elimination, *Proc. Programming Language Design and Implementation* (*PLDI'94*), pp.159–170, ACM (1994).
6) Chow, F.C. and Hennessy, J.L.: The Priority-Based Coloring Approach to Register Allocation, *ACM Trans. Prog. Lang. Syst.*, Vol.12, No.4, pp.501–536 (1990).
7) Click, C.: Global Code Motion Global Value Numbering, *Proc. Programming Language Design and Implementation* (*PLDI'95*), pp.246–257, ACM (1995).
8) Dhamdhere, D.M.: Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.2, pp.291–294 (1991).
9) Dhamdhere, D.M. and Patil, H.: An Elimination Algorithm for Bidirectional Data Flow Problems Using Edge Placement, *ACM Trans. Prog. Lang. Syst.*, Vol.15, No.2, pp.321–336 (1993).
10) Dhamdhere, D.M., Rosen, B.K. and Zadeck, F.K.: How to Analyze Large Programs Efficiently and Informatively, *Proc. Programming Language Design and Implementation* (*PLDI'92*), pp.212–223, ACM (1992).
11) Chow, S.C.F., Kennedy, R., Liu, S.M. and Lo, R.: A New Algorithm for Partial Reundancy Elimination based on SSA Form, *Proc. Programming Language Design and Implementation* (*PLDI'97*), pp.273–286, ACM (1997).
12) Fraser, C. and Hanson, D.: *A Retargetable C Compiler: Design and Implementation*, Addison Wesley (1995).
13) Khedker, U.P. and Dhamdhere, D.M.: A Generalized Theory of Bit Vector Data Flow Analysis, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.5, pp.1472–1511 (1994).
14) Knoop, J., Rüthing, O. and Steffen, B.: Lazy Code Motion, *Proc. Programming Language Design and Implementation* (*PLDI'92*), pp.224–234, ACM (1992).
15) Knoop, J., Rüthing, O. and Steffen, B.: Optimal Code Motion: Theory and Practice, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.4, pp.1117–1155 (1994).
16) Knoop, J., Rüthing, O. and Steffen, B.: The Power of Assignment Motion, *Proc. Programming Language Design and Implementation* (*PLDI'95*), pp.233–245, ACM (1995).
17) Lo, R., Chow, F., Kennedy, R., Liu, S. and Tu, P.: Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores, *Proc. Programming Language Design and Implementation* (*PLDI'98*), pp.26–37, ACM (1998).
18) Morel, E. and Renvoise, C.: Global Optimization by Supperssion of Partial Redundancies, *Comm. ACM*, Vol.22, No.2, pp.96–103 (1979).
19) Muchnick, S.S.: *Advanced Compiler Design Implementation*, Morgan Kaufmann (1997).
20) Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Global Value Numbers and Redundant Computations, *Proc. Principles of Programming Languages* (*POPL'88*), pp.12–27, ACM (1988).
21) Chow, F.C., Chan, S., Liu, S., Lo, R. and Streich, M.: Effective Representation of Aliases and Indirect Memory Operations in SSA Form, *Proc. Int. Compiler Construction* (*CC'96*), LNCS, Berlin, Springer-Verlag (1996).
22) Steffen, B., Knoop, J. and Rüthing, O.: The Value Flow Graph: A Program Representation for Optimal Program Transformations, *Proc. Int. European Symposium on Programming* (*ESOP'90*), Copenhagen, Denmark, pp.389–405, Springer-Verlag (1990).
23) Takimoto, M. and Harada, K.: Partial Dead Code Elimination Using Extended Value Graph, *Proc. Int. Static Analysis Symposium* (*SAS'99*), Vol.1694 of LNCS, Venice, Springer-Verlag (1999).
24) Takimoto, M. and Harada, K.: Eliminating May-aliases, *Proc. Int. Advances in Infrastructure for Electronic Business, Science, and Education on the Internet* (*SSGRR'01*), L'Aquila (2001).
http://www.ssgrr.it/en/ssgrr2001/papers.htm
25) 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).
26) 滝本宗宏，原田賢一：拡張値グラフを用いた部分無効コード除去法，情報処理学会論文誌，Vol.41, No.1, pp.46–58 (2000).

(平成 14 年 2 月 19 日受付)
(平成 14 年 5 月 16 日採録)

滝本　宗宏（正会員）

1967 年生．1994 年慶応義塾大学大学院理工学研究科計算機科学専攻修士課程修了．現在，東京理科大学理工学部情報科学科助手，プログラミング言語およびその処理系に興味を持つ．ACM，ソフトウェア科学会会員．

原田　賢一（正会員）

1940 年生．1966 年慶応義塾大学大学院工学研究科管理工学専攻修士課程修了．1967 年同大学工学部助手．1970〜1989 年同大学情報科学研究所助手，専任講師，助教授，教授．1989 年 4 月より同大学理工学部計測工学科教授．1996 年 4 月より同学部情報工学科教授．この間，1973〜1975 年米国メリーランド大学訪問研究員．工学博士．ソフトウェア工学，プログラミング言語およびその処理系の研究に従事．ACM，IEEE，ソフトウェア科学会会員．