部分冗長コードの多重ループ外への一挙移動方式

佐瀬拓矢[†,☆] 鈴木貢[††] 渡邊坦[††]

オブジェクトコード最適化では，ループ不変コードのループ外への移動が効果的である．多重ループの場合，内側のループから外側のループへと逐次移動してゆく方法では，移動に伴う新たな移動条件判定やデータフロー情報の更新等により，コンパイル時間が長くなる．部分冗長コード抑制アルゴリズムには，ループ不変コードを移動可能な最も外側のループの入口へ一挙に移動できるものがあるが，削除できる部分冗長コードがかなり限定されたり，コード量がかなり増大する，あるいは実現が容易でないという問題がある．

本論文は，多重ループの内側から外側へ，部分冗長コードを一挙に移動する新たな方法を提案する．これは，部分冗長性削除の考え方を参考にして，多重ループの効率化の解決方法を示したもので，従来の方法では削除できないコードも，新しい移動条件の設定とグラフ構造の変形により削除できる．その変形は複雑なものではなく，データフロー情報の何段もの更新を必要としない．また，移動条件を論理演算式で表すことにより，コンパイル時間も短く，実装も容易なものとして実現した．

Motion of Partially Redundant Codes to Outside of Nested Loops in One Step

TAKUYA SAZE,[†] MITSUGU SUZUKI[††] and TAN WATANABE[††]

Removal of loop invariant codes from loop inside is effective in optimizing object code. For nested loops, stepwise movement of invariant code from inner loop to outer loop takes much compile time in recomputing conditions for movement and updating dataflow information at each step. Some partial redundancy elimination methods can move loop invariants from inner loop to most possible outside point of nested loop in one step, however, they have such drawbacks as either some invariants can not be moved or code size increases much or implementation is not easy.

This paper introduces a new method that can move partially redundant codes as well as loop invariants from inner loop to outside of nested loop in one step. It is an application of the concept of partial redundancy elimination to code optimization of nested loops and it can move some loop invariants that can not be moved by other method by introduction of new criterion for code motion and by simple control graph restructuring. It does not require repeated update of dataflow information. The moving criterion and destination point can be computed by solving logical equations. It can be easily implemented and does not require much compile time.

1. はじめに

コンパイラにとって重要なことは，実行効率の良いオブジェクトコードの生成[1]である．それを実現するオブジェクト最適化では，プログラム実行時間の大部分を占めるループの効率向上[2]が非常に有効である．ループの効率化には，ループ帰納変数の効率化やループ展開なども有効であるが，ループ不変コードのループ外への移動が効果の大きい方法なので，それをうまく実現することを主目的として研究した．よく知られているループ不変コードの移動方法[1),3)]は，一重ループの外側に移動する方法として定式化されている．これを多重ループに適用するには，内側のループから外側のループへと順次行なえばよい．すなわち，不変コードはそれを含むループの外へ移動し，それをまた外側のループへと，段々に移動していく．この方法では，移動に伴う新たな移動条件判定やデータフロー情報の更新等により，コンパイル時間が長くなる．他の方法として，部分冗長コードを移動可能な最も外側のループの入口へ一挙に移動できる Morel と

† 電気通信大学大学院情報工学専攻
Computer Science, University of Electro-Communications
☆ 現在，富士通株式会社
Presently with Fujitsu Limited
†† 電気通信大学情報工学科
University of Electro-Communications

Renvoise の部分冗長コード抑制アルゴリズム [4] があり，これは部分冗長コードの一種としてのループ不変コードにも適用できる．しかし，これでは制御フローグラフの構造を変えないので，削除できない部分冗長コードが残る．一方，コードを複写しグラフを再構築する方法 [5] では，ほとんどの部分冗長コードを削除できるが，コード量がかなり増大し，実現もあまり容易でない．

本論文では，部分冗長コードを多重ループの外へ一挙に移動する方法として，コンパイル時間が短く，実行効率が良いオブジェクトコードを生成し，かつ，実現が容易であるものを，新たに提案し，その実装結果を報告する．

2. 従来の部分冗長コード移動の手法

あるコードが部分冗長であるとは，そのコードが位置する点に至る経路群の少なくとも1つに，そのコードで使うオペランドの変更がなく，それ以前に計算された値がその点で利用できることである．ループ内で不変なコードも部分冗長コードの一種である．したがって，部分冗長コードを効率化する手法の案出は，実行効率のよいオブジェクトの生成に欠かせない．

ループ不変コードをループ外に逐次移動する方法 [1),3)] に対し，Morel 達による部分冗長コード削除 PRE (Partial Redundancy Elimination) の方法 [4] は，部分冗長コードを一挙に最も外側のループの外へ移動することもできる．これは，ループ不変コードを移動するだけではなく，大域的な共通部分式も削除できる．しかしながら，この方法はプログラムのグラフ構造を変えずにコード移動を行なうため，多くの部分冗長コードが残ってしまう．例えば，図1の `x + y` は明らかにループ不変であるが，このプログラムでは 1→2→EX の経路上に `x + y` の計算がないために，Morel 達の PRE ではループの外に移動することができない．コードを複写しグラフを再構築して処理する方法 [5] では，部分冗長コードをほぼ全部削除できるものの，コード量がかなり増大し，処理も複雑である．

本論文で提案する方式では，部分冗長コードを，移動可能な最も外側のループの外へ一挙に出すことを目標とした．それにより，ループ不変コードのループ外移動方法で生じるコンパイル時間を縮小できるとともに，Morel 達の PRE 同様，大域的な共通部分式も削除できる．また，Morel 達の PRE では残るコードも，本方式では必要があればグラフ構造を変えることにより削除できる．しかもその再構築は複雑なものではなく，コンパイル時間を大きく増大させるものでもない．

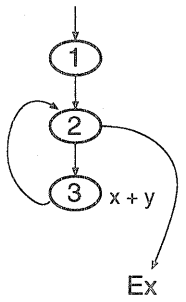


図1 サンプル1
Fig. 1 Sample 1.

3. 本方式のアルゴリズムの概要

提案する方式のアルゴリズムの骨子は次の通りである．

(1) プログラムのグラフ構造を調べ，必要があれば再構築する．
(2) 基本ブロックのデータフロー情報に関する，局所的・大域的プロパティを求める．局所的な共通部分式を削除する．
(3) 部分冗長コードのオペランドが変更されていない基本ブロックでは，その部分冗長コードを移動できるかもしれないとして，コードが削除される可能性のあるブロック群を求め，それらの先行ブロック群へその情報を伝える．
(4) 新しいコードを挿入すると判定されたブロックに，コード挿入を行なう．それにより部分冗長コードは完全冗長となり，削除される．

4. グラフ再構築

必要に応じて新たなブロックを挿入する条件について説明する．

最前ヘッダ (*Most preheader*):
ある多重ループ n における，最外ループの入口ブロック（ヘッダ）h の先行ブロック p が1つであって，それは最外ループに含まれないブロックであれば，そのブロック p を多重ループ n の最前ヘッダと呼ぶ．次の2つのどちらかに当てはまる場合，多重ループ n に最前ヘッダ mp を新たに挿入する．挿入する最前ヘッダ mp の後続ブロックはヘッダ h のみであり，mp の先行ブロックは，元のヘッダ h のすべての先行ブロックである．(この最前ヘッダの説明におけるヘッダの先行ブロックからは，多重ループに含まれるブロックは除外する．)

- ヘッダ h の先行ブロックの数が1つではない．

- ヘッダ h の先行ブロック p の後続ブロックの数が1つではない．

ループの出口ブロック (*Loop exit block*):
あるループ n の中のブロック t を先行ブロックとして持ち，かつそのループ n に含まれていないブロックをループ n の出口ブロック ex と呼ぶ．これはループ n から出る1つの経路として，t の所から ex へ出るものがあることを表す．(出口ブロック ex を後続ブロックとして持つループ内ブロック t を，ループの前出口ブロックと呼ぶ)．次の2つのどちらかに当てはまる場合，出口ブロックを新たに挿入する．

- ループ n の出口ブロックが，ループ m の入口ブロックにもなっている場合，ループ n の出口ブロックを新たに挿入する．
- あるループの出口ブロックが複数のループの出口となっていて，かつ，複数の前出口ブロックを有する場合，それぞれのループの出口ブロックを新たに挿入する．

新しい出口ブロックの後続ブロックは，かつての出口ブロックのみであり，先行ブロックはそれぞれのループの前出口ブロックのみである．出口ブロックは，ループの外側から内側へコードが移動するのを防ぐ役割を持っている．

出口ブロックであり，かつ最前ヘッダでもあるブロックは，最前ヘッダとして処理する．

5. 基本ブロックの論理代数プロパティ

ある演算を表すコードに対し，まずそのコードが局所的に利用可能であるか等の解析をし，次にそれらを元に大域的に利用可能であるか等の解析を行なう．そこで得られたデータから，部分冗長コード移動のためのデータを求める．これらは全て基本ブロックごとにその演算コードに対するプロパティとして表現し，論理代数によってその値を求める．l 個の演算があるとすると，1つのプロパティに対して長さ l のビットベクトルを各基本ブロックごとに持たせ，その第 i ビットで第 i 演算のプロパティが何であるかを示す．プロパティ群の詳細な説明を次に示す．ここで述べる局所的 大域的プロパティは，Morel 達の論文[4]にならっているので，記号もそれにならう．

局所的プロパティ 基本ブロックごとに個別に計算できるプロパティを次に示す．

- $TRANSP_i$:
  ある演算 e のオペランドの値が，ブロック i における命令の実行によって変更されないとき，$TRANSP_i[e] = TRUE$ とし，e がブロック i で透明であることを表す．(文献3) の *e-kill* の否定に相当)
- $COMP_i$:
  ある演算 e の最後の計算の後に現れる命令が，ブロック i でそのオペランドの値を変更しないとき $COMP_i[e] = TRUE$ とする．(文献3) の *e-gen* に相当)
- $ANTLOC_i$:
  ある演算 e の最初の計算の前に現れるどの命令もブロック i でそのオペランドの値を変更しないとき $ANTLOC_i[e] = TRUE$ とし，その値がブロック i の入り口で局所的に予期可能であることを表す．

大域的プロパティ 異なる基本ブロック間で相互依存するプロパティとその計算方法を次に示す．ここで，$Pred(i)$ は i の先行ブロック，$Succ(i)$ は i の後続ブロックのことを意味する．

- *Availability System*
  $AVIN_i$: ある演算 e がブロック i の入口で利用可能なとき $AVIN_i[e] = TRUE$. (文献3) の *avail-in* に相当)
  $AVOUT_i$: ある演算 e がブロック i の出口で利用可能なとき $AVOUT_i[e] = TRUE$. (文献3) の *avail-out* に相当)

$$AVIN_i = \begin{cases} FALSE & \text{:if } i \text{ is the subprogram entry block} \\ \prod_{j \in Pred(i)} AVOUT_j & \text{:otherwise} \end{cases}$$

$$AVOUT_i = COMP_i + TRANSP_i \cdot AVIN_i$$

- *Anticipability System*
  $ANTOUT_i$: ある演算 e がブロック i の出口で予期可能（すべての後続ブロックの入り口で予期可能）なとき $ANTOUT_i[e] = TRUE$.
  $ANTIN_i$: ある演算 e がブロック i の入口で予期可能なとき $ANTIN_i[e] = TRUE$.

$$ANTOUT_i = \begin{cases} FALSE & \text{:if } i \text{ is the subprogram exit block} \\ \prod_{j \in Succ(i)} ANTIN_j & \text{:otherwise} \end{cases}$$

$$ANTIN_i = ANTLOC_i + TRANSP_i \cdot ANTOUT_i$$

- *Partial Availability System*
  $PAVIN_i$: ある演算 e がブロック i の入口で部分的に利用可能なとき $PAVIN_i[e] = TRUE$.
  $PAVOUT_i$: ある演算 e がブロック i の出口で部分的に利用可能なとき $PAVOUT_i[e] = TRUE$.

$$PAVIN_i = \begin{cases} FALSE \\ \quad :\text{if } i \text{ is the subprogram entry block} \\ \sum_{j \in Pred(i)} PAVOUT_j \\ \quad :\text{otherwise} \end{cases}$$

$$PAVOUT_i = COMP_i + TRANSP_i \cdot PAVIN_i$$

部分冗長コード抑制プロパティ 部分冗長コード移動のために本論文で新たに導入したプロパティを次に示す．それらの求め方はあとでまとめて示す．

- $DELETABLE_i$:
  ブロック i において，ある演算 e が削除可能なとき，$DELETABLE_i[e] = TRUE$.
- *PRExpInfo (Partial Redundant Expression Information)*
  $PRExpInfoTP_i$:
  ブロック i において，ある演算 e がブロック i，または後続ブロックに削除可能な部分冗長計算として存在するとき $PRExpInfoTP_i[e] = TRUE$.
  (ここで，TP_i とは *To Predecessor*，つまり i の先行ブロックへ，ということを意味する.)
  $PRExoInfoFS_i$:
  ブロック i において，ある演算 e が後の経路に削除可能な部分冗長計算として存在するとき $PRExoInfoFS_i[e] = TRUE$.(ここで，FS_i とは，*From Successor*，つまり i の後続ブロックから，ということを意味する.)
- $INSERT_i$:
  ブロック i において，ある演算 e が挿入できるとき $INSERT_i[e] = TRUE$. このとき e はブロック i の末尾に挿入する.
- $DELETE_i$:
  ブロック i において，ある演算 e が削除できるとき $DELETE_i[e] = TRUE$. このときブロック i に現れる最初の e の計算を削除する.
- $OpSO_i$(Operand Set Only):
  このプロパティは，ループの出口ブロック i でのみ使用する．この出口ブロックを出口とするループ中で，ある演算 e は計算されず，そのオペランドの変更のみが行なわれているとき $OpSO_i[e] = TRUE$.

論理演算式 上記のプロパティの間には，次のような関係が成り立つ．ここで，$Loop(n)$ はあるループ n 中のブロックのことを意味する.

$$DELETABLE_i = \begin{cases} \neg OpSO_i \cdot ANTLOC_i \cdot PAVIN_i \\ \quad :\text{if } i \text{ is loop exit block} \\ ANTLOC_i \cdot PAVIN_i \\ \quad :\text{otherwise except most prehead block} \end{cases}$$

$$PRExpInfoTP_i = \begin{cases} FALSE \\ \quad :\text{if } i \text{ is most preheader block} \\ DELETABLE_i + \neg OpSO \\ \quad \cdot PRExpInfoFS_i \\ \quad :\text{if } i \text{ is loop exit block} \\ DELETABLE_i + PRExpInfoFS_i \\ \quad \cdot TRANSP_i \cdot \neg ANTLOC_i \\ \quad :\text{otherwise} \end{cases}$$

$$PRExpInfoFS_i = \sum_{j \in Succ(i)} PRExpInfoTP_j$$

$$INSERT_i = \begin{cases} PRExpInfoFS_i \cdot \neg COMP_i \\ \quad \cdot(\neg TRANSP_i + TRANSP_i \cdot \neg AVIN_i) \\ \quad :\text{if } i \text{ is most preheader block} \\ \quad\quad \text{or the subprogram entry block} \\ PRExpInfoFS_i \cdot \neg COMP_i \\ \quad \cdot(\neg TRANSP_i + TRANSP_i \cdot OpSO) \\ \quad :\text{if } i \text{ is loop exit block} \\ PRExpFS_i \cdot \neg TRANSP_i \cdot \neg COMP_i \\ \quad :\text{otherwise} \end{cases}$$

$$DELETE_i = \begin{cases} ANTLOC_i \cdot AVIN_i \\ \quad :\text{if } i \text{ is most preheader block} \\ DELETABLE_i \\ \quad :\text{otherwise} \end{cases}$$

$$OpSO = \prod_{k \in Loop(n)} \neg ANTLOC_k \cdot \neg COMP_k$$

:only for the loop exit block of $Loop(n)$

6. 部分冗長コードの移動

ループ内の演算はできる限り外側のループ，あるいはループの外で行なうことにし，そのコードの挿入可能な位置を求める．その位置は，基本的に，最前ヘッ

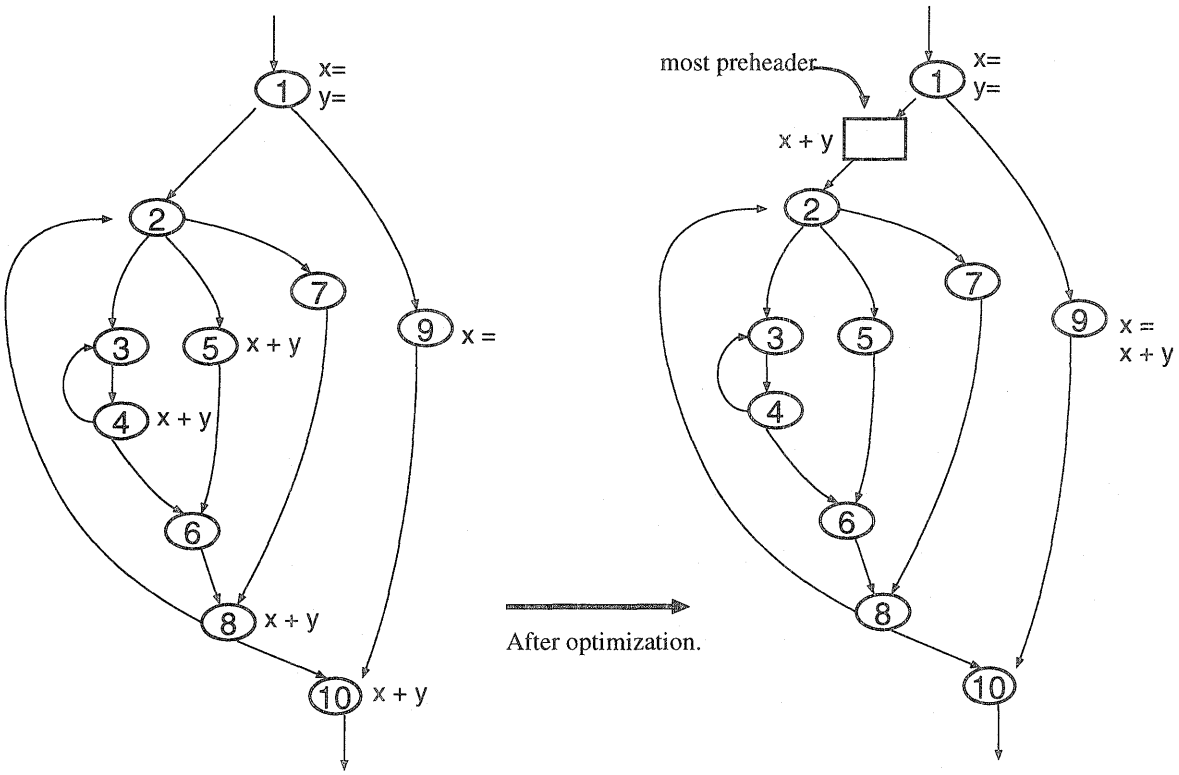


図 2 最適化の例
Fig. 2 Example of Optimization.

ダ，あるいはオペランド変更直後とする．具体的には，ある基本ブロックの出口に，その後の経路に削除可能なコードがあるという情報が伝わり，かつそのブロックが最前ヘッダであるか，あるいはオペランド変更があれば，そこを挿入点とする．上記の論理演算式ではそのブロックは *INSERT* で示される．これで求められたブロックの末尾へのコード挿入により，部分冗長な演算が完全冗長となり，*DELETE* で示されるコードは削除できる．ある演算 e のオペランドがループの中で設定されているが e 自体はループ内で設定されていない場合，*OpSO* を見て，e の演算はオペランド設定の直後までは遡らず，そのループの出口ブロックでなされる．

これらの処理は，ループ内の演算に限らず，すべての部分冗長コードを対象とする．

論理演算式による表現は，ビットベクトルにより複数の演算を同時に処理できるので，オペランドの定義や使用の情報参照や条件チェック等の処理を手続き的に書くよりも，処理時間が短く，実装も容易である．

適用例を図 2 に示す．本方式では，ブロック 2 の前に最前ヘッダを追加してそこに x + y を挿入すると，ブロック 4, 5, 8 の x + y は完全冗長となって削除される．ブロック 9 では，オペランド x の値設定の後に x + y の再計算が挿入され，ブロック 10 の x + y は削除される．Morel 達の方法では，ブロック 9 を通るパスではその演算が不要なので，ブロック 4, 5, 8 の x + y はブロック 2 には移動できても，ブロック 1 には移動できない．ブロック 1 に移動すると，ブロック 9 を通る経路では x の設定によってそれが無効になってしまうからである．すなわち，ブロック 2 で始まるループの外に出すことはできない．本方式では，これに対し，最前ヘッダを追加してそこに移動するので，最外ループの外に出すことができる．

7. コード挿入における問題と解決

6 節で述べたコード移動では，次のような問題が生じる．例えば，図 3 の左側のグラフに，本方式を適用すると，右側のグラフになる．部分コード $e1$ を含むコード $e2$ が，ブロック 1, 4 で $e1$ の値を利用できないにもかかわらず，$e2$ を実行してしまう．この問題は，Morel 達の方法でも生じるもので，Drechsler 達[6] は，各基本ブロック間への新しいブロックの挿入によるグラフ構造の変形と，移動条件の改良によってこの問題を解決した．我々は，挿入するコードのオペランド（部

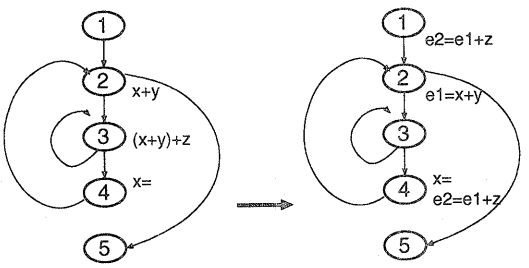


図 3 部分冗長式移動に伴う問題
Fig. 3 Problem with Motion of Partial Redundancy.

分コード）の値が利用可能かどうかを検査し，利用できない場合にはオペランドのコードを挿入コードの直前に挿入するという方法を案出した．この方法では，ブロックの挿入を行なわないので，Drechsler 達の方法より低い処理コストで実現可能である．検査の条件は，次の論理演算式で表される．

$$OpCHECK_i = INSERT_i + COMP_i + AVIN_i \cdot TRANSP_i$$

挿入コードのオペランドの $OpCHECK$ が $TRUE$ のときは，オペランドのコードを挿入しない．この処理を，6 で述べた手順に織り込んで，コードの移動を行なう．

8. 実装したコンパイラの構成

本研究で開発中のコンパイラは，複数の言語と複数の機種の各組み合わせに対し，統一的な形で構築するため，図 4 のようにフロント・エンド，中間処理部，バック・エンドの三つの部分から成っている [7]．

フロント・エンドはプログラム言語に依存する言語解析処理部であり，ソースコードを中間表現に変える．中間表現は，木構造中間語を経由して，中間処理部で，言語・機種に依存しない抽象レジスタ型中間語 ParmCode (Parallel Abstract Register Machine Code) へ変換する．ここで中間語向け最適化処理やレジスタ割り付け等も行なう．バック・エンドでは ParmCode からオブジェクトコードを生成する．ここで機械のアーキテクチャの特徴を利用した最適化処理を行なうこともできる．このような 3 つの部分を構成する言語対応モジュールや機種対応モジュール，言語・機種非依存モジュールを組み合わせ可能とすることにより，多言語・多機種への対応を容易にする．

本方式の実装は，中間処理部の ParmCode について行ない，言語・機種に非依存な最適化処理として構築している．冗長なコードは，ソースコードに含まれるものばかりでなく，配列要素などに対して ParmCode に変換する過程で生成される変数のアクセスコードなどとしても，多数含まれる．それらの大部分はコード最適化の過程で削除される．

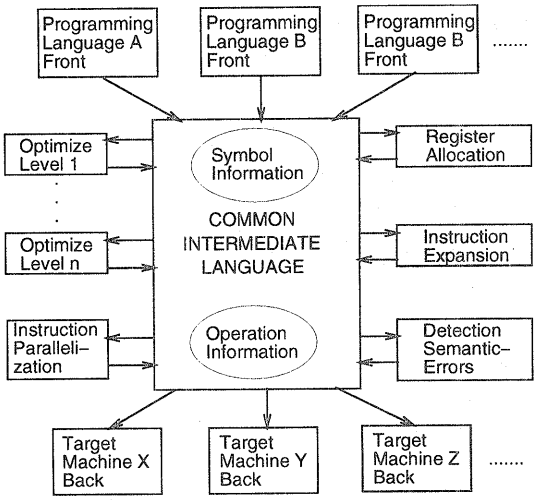


図 4 コンパイラの構成
Fig. 4 Structure of the Compiler System.

9. 性能評価

実装した系に，多重ループ構造のプログラムを与えて，性能の評価を行なった．使用した計算機は Sun Microsystems 社の SparcStation 5（MicroSPARCII 70MHz，主記憶 64MB）と，SunOS4.1.4 である．サンプルプログラムは，A_i を nrow[i] × nrow[$i+1$] 型の行列として ($i = 0, 1, \ldots, N-1$)，nrow[i] ($i = 0, 1, \ldots, N$) を与えると，積 $A_0 A_1 \ldots A_{N-1}$ の計算のための最少乗算数（コスト）を求めていく動的計画法，そして整列アルゴリズムであるヒープソートの 2 つである．

比較対象となる最適化技法として，ループ不変コードのループ外への逐次的移動方法と Morel 達の PRE を実装して評価した．ただし，ループ不変コードのループ外移動方法を適用する際には，前もって局所的・大域的な共通部分式削除を行なった．局所的な共通部分式削除を行なう理由は，1 つの基本ブロック内では同じ計算を 1 回だけやることにして処理を単純化するためである．大域的な共通部分式削除を行なう理由は，4 節で述べたように，本方式と Morel 達の PRE では大域的な共通部分式も削除できるからである．ループ不変コードのループ外移動方法は，表では Loop と記した．

最適化処理時間 他の最適化技法との最適化処理時間

表 1 他の技法との最適化処理時間の比
Table 1 Optimization-Time Ratio with Other Methods.

	Loop	PRE	本方式
動的計画法	1.00	0.46	0.45
ヒープソート	1.00	0.51	0.52

表 2 他の技法との実行時間比
Table 2 Execution-Time Ratio with Other Methods.

	最適化なし	Loop	PRE	本方式
動的計画法	1.00	0.47	0.60	0.45
ヒープソート	1.00	0.84	0.78	0.80

比を**表 1** に示す．Loop を 1.00 としているのは，ループ不変コードのループ外移動方法でかかる最適化処理時間の削減を目標としたためである．

表 1 の結果から，部分冗長コードを一挙に多重ループの外に出す本方式と PRE が，何度かのステップを経るループ不変コードのループ外移動方法よりもはるかに最適化処理時間が少ないのが分かる．また，本方式は必要であればグラフ構造を再構築するが，グラフ構造を変えない PRE との最適化処理時間を比べてみると，その再構築に要する時間は大きなものではないことが分かる．このことから，本方式はコンパイル時間を大きく削減するものであると言える．

実行時間 他の最適化技法との実行時間比を**表 2** に示す．最適化を行なわないオブジェクトに対して，どの程度実行効率が向上したかを見た．

表 2 の結果を見ると，本方式では Morel 達の PRE が削除できない部分冗長コードを削除したことにより，動的計画法では実行効率が向上している．だがヒープソートでは，それほど差はないとはいえ，PRE が優っている．動的計画法のプログラムは分岐が少ないものであるが，ヒープソートは分岐を多数含んでいる．表 2 の結果は，分岐の多いプログラムでは，移動する部分冗長コードをもっと絞りこむ必要があることを示している．

10. 考　　察

本方式は，部分冗長コードの移動可能先を求めるためのプロパティ（*PRExpInfo* 等）を導入し，多重ループの外へ一挙に移動することにより，通常のループ不変コードのループ外移動方法に比べ，コンパイル時間ではサンプルプログラムにおいては 50%前後のコスト削減を実現している．ループ不変コードのループ外移動方法はプログラムの大きさだけでなく，ループネストの深さに比例する時間を必要とするが，本方式でのコンパイル時間はプログラムの大きさだけによるからである．プログラムが多重ループ構造であればあるほど，本方式は力を発揮するであろう．また，実装は，論理演算式を使うことにより，ループ不変コードのループ外移動を手続きで記述するものよりはるかに容易であった．

Morel 達の PRE と比較すると，コンパイル時間はほぼ同等であるが，実行効率は PRE では削除できない部分冗長コードを削除したことにより，サンプルプログラムによっては約 25%向上している．ただし，常に本方式が PRE に優るわけではなく，プログラムによっては PRE による実行効率向上が上の場合もある．本方式は，ループ外に出した演算がループ本体を 1 回も通らない場合には無駄になる等の場合があるためである．

これへの対策としては，while 型ループを条件節を前に追加した repeat until 型ループに変えるとか，実行頻度の低い経路では移動を抑制するなどして，よりきめ細かなコード移動を検討する必要がある．そのためには，その条件判定の処理の追加などが必要となるが，そのようなきめ細かな対応は，本方式のような一律な方法によってあげられる適用範囲の広い方法に比べ，労力投入に対する効果の比率は次第に低くなると予想される．

Bodik 達の方法[5]とは直接比較できていないが，彼らの方法は部分冗長性を完全になくすことを目的としているので，実行効率はおそらく本方式より良いと思われる．しかし，それはコードの巻き上げを阻害する領域を求め，それを経路別に複写することによって移動を可能にする方法なので，コード量が約 30%（場合によっては約 2 倍に）増加し，処理内容も本方式より複雑である．本方式では，追加するブロックはループの最前ヘッダと出口ブロックのみであり，コード量は減ることが多く増えることは少ない．

本方式の特長は，多重ループの効率化に焦点を絞ることにより，Morel 達の方式を参考にして，部分冗長コードを多重ループの外へ一挙に移動可能とし，実装容易な方法で大きな効果を上げられることを示した点にある．最高性能ではなくても，実装容易でかつ効果の大きい方法は，工学的には有用と思われる．

今後の課題として，コード移動によって発生する新たな部分冗長性の抑制や，コード量増加の抑制なども更なる性能向上のために考えなければならないであろう．また，文献 5) などとの定量的な比較や，多数の例題による評価も課題として残されている．

(平成 11 年 3 月 10 日受付)
(平成 11 年 4 月 28 日採録)

11. 謝　辞

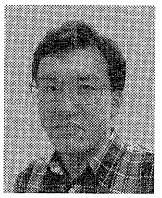
本論文に対して多くの貴重なご意見を賜った査読者の方々，ならびに 1999 年 3 月のプログラミング研究会でたいへん有益な討論を頂いた方々に深く感謝致します．

参 考 文 献

1) Aho, A. V., Sethi, R. and Ullman, J. D.: Compilers Principles, Techniques, and Tools, *Addison-Wesley publishing company (1986)*.
2) Bacon, D. F., Graham, S. L. and Sharp, O. J.: Compiler Transformations for High-Performance Computing, *ACM Computing Surveys*, Vol. 26, No. 4, pp. 345–420 (1994).
3) 佐々政孝: プログラミング言語処理系, 岩波書店, 1989.
4) Morel, E. and Renvoise, C.: Global Optimization by Suppression of Partial Redundancies, *Comm. of the ACM*, Vol. 22, No. 2, pp. 96–103(1979).
5) Bodik, R., Gupta, R. and Soffa, M. L.: Complete Removal of Redundant Expressions, *Proc. PLDI'98*, pp. 1–14 (1998).
6) Drechsler, K. H. and Stadel, M. P.: A Solution to a Problem with Morel and Renvoise's Global Optimization, *ACM TOPLAS*, Vol. 10, No. 4, pp. 635–640 (1988).
7) 渡邊 坦, 鈴木 貢: コンパイラ・エンジニアリング, 第 38 回プログラミング・シンポジウム報告集, 情報処理学会, pp. 135–142 (1997).

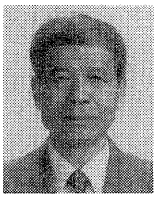
佐瀬　拓矢

1972 年生．1997 年電気通信大学電気通信学部情報工学科卒．1999 年電気通信大学大学院電気通信学研究科情報工学専攻修士課程修了．同年富士通株式会社入社．コンパイラ，計算機アーキテクチャに興味を持つ．



鈴木　貢（正会員）

電気通信大学情報工学科助手．記憶管理方式，並列アルゴリズム，プログラミング言語処理系 等に興味を持つ．ACM，電子情報通信学会，日本ソフトウエア科学会 各会員．



渡邊　坦（正会員）

1962 年 京都大学理学部数学科卒．日本 IBM（株），（株）日立制作所中央研究所，同システム開発研究所を経て，1994 年より電気通信大学情報工学科教授．工学博士．プログラミング言語とプログラミング・ツール，各種言語処理系の開発を行なってきた．現在は，コンパイラの研究・開発を主要テーマとしている．著書「コンパイラの仕組み」（朝倉書店）．日本ソフトウエア科学会，ACM，IEEE 各会員．