静的単一代入形式からの逆変換アルゴリズムの比較と評価

伊藤 陽[†] 小濱真樹[††] 佐々政孝[†]

SSA 形式（静的単一代入形式）は，ϕ 関数という仮想的な関数を用いることで，変数の定義を字面上 1 カ所だけになるように表した中間表現である．しかし，SSA 形式を用いたプログラム中に現れる ϕ 関数が仮想的であることから，SSA 形式から直接目的コードを出すことはできない．そこで，目的コードを出す前段階として，ϕ 関数を消去して通常形式に戻すことが必要である．これを SSA 逆変換と呼ぶ．SSA 逆変換には主なアルゴリズムが 2 つある．以前から用いられている Briggs らのアルゴリズム，およびこれとは異なるアプローチによる Sreedhar らのアルゴリズムである．SSA 逆変換アルゴリズムはそれぞれ異なった特徴を持つが，これらを実際に比較した研究はほとんどなされていない．このため，SSA 形式を最適化コンパイラに採用するにあたって，どの SSA 逆変換を用いるかの選択の指標となるものがなかった．そこで本研究では，Briggs らと Sreedhar らのアルゴリズムの長所と短所を明確にした．また，Briggs らのアルゴリズムに対する改良案を提案した．さらに，Briggs らのアルゴリズムとその改良案，Sreedhar らのアルゴリズムの 3 つを実装し，同一のコンパイラ上で，SPEC ベンチマークを用いて種々の条件を変えながら比較した．本研究により，Briggs らの方法ではコアレッシングすることのできないコピー文が数多く挿入されること，実証的には Sreedhar らの方法が最も優れていること，が判明した．

Comparison and Evaluation of Reverse Translation Algorithms for Static Single Assignment Form

Yo Ito,[†] Masaki Kohama[††] and Masataka Sassa[†]

The SSA form (static single assignment form) is an intermediate representation where the definition of each variable appears only once in the program by using a hypothetical function called ϕ-function. However, since the ϕ-functions appearing in the program using the SSA form are non-executable, the target code cannot be directly generated from the SSA form. Therefore, it is necessary to delete the ϕ-function and put the SSA form back to the normal form before generating the target code. This conversion is called SSA reverse translation'. There are two major SSA reverse translation algorithms. One is the method by Briggs et al., which is used from relatively early times. The other is the method by Sreedhar et al., which is based on a completely different approach. So far, there has been almost no research that actually compares these SSA reverse translation algorithms, although each algorithm has different characteristic features. Therefore, there have been no criteria as to which SSA reverse translation algorithm should be selected when the SSA form is used in an optimizing compiler. In this paper, we clarify merits and weak points of algorithms by Briggs and Sreedhar. We also propose an improvement of the algorithm by Briggs. Then, we implement algorithms by Briggs and its improvement and the algorithm by Sreedhar on the same compiler, and compare the three by changing the various environments using the SPEC benchmarks. The experiment has shown that the method by Briggs inserts a lot of copy statements that cannot be coalesced. Empirically, the method by Sreedhar is the most efficient.

1. はじめに

SSA 形式（静的単一代入形式）は，ϕ 関数という仮想的な関数を用いることで，変数の定義を字面上 1 カ所だけになるように表した中間表現である．SSA 形式を用いると，最適化の実現容易性と最適化の実行効率が向上するといわれている．このため多くの最適化コンパイラ[6),7),9)~11),18)] が SSA 形式を採用（一部試験採用）している．

しかし，SSA 形式を用いたプログラム中に現れる ϕ 関数が仮想的であることから，SSA 形式から直接アセンブリ言語や機械語などの目的コードを出すことはできない．よって，目的コードを出す前段階として，ϕ

† 東京工業大学情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology
†† 富士写真フィルム
Fuji Photo Film Co., Ltd.

関数を消去して通常形式に戻すことが必要である．これを **SSA 逆変換**と呼ぶ．

SSA 逆変換は，基本的には SSA 形式への変換と逆の作業を行えばよい．しかし，SSA 形式を用いた中間表現から目的コードへの変換の間に行われる最適化によって，コードの移動やコピー伝播が行われると，素朴な処理では実現できず，注意深く設計したアルゴリズムが必要である．

実際，Cytron らの提案した SSA 逆変換アルゴリズム[5] では，変換前と変換後のプログラムの意味が変わってしまう場合があることが知られている[1]．

実用的なアルゴリズムとしては Briggs らの提案した方法[1] が古くから用いられている．また，Sreedhar らはこれとは異なるアプローチによるアルゴリズム[19] を提案したが，Briggs らの方法と比べて歴史的に新しいためか，採用されている事例が少ないようである．SSA 形式を用いたコンパイラで，Briggs らの方法を用いたものには Marmot[6]，Machine SUIF[9]，Scale[18] などがあり，Sreedhar らの方法を用いたものには OpenJIT[12),15)] などがある．

Sreedhar らの方法では ϕ 関数に対し，ある種のコアレッシング（合併）に類似した処理をすることで SSA 逆変換を行う．また，Briggs らの方法は ϕ 関数をコピー文で代用することによって SSA 逆変換を行う．

前者のアプローチでは 1 つの変数の生存区間が長くなる．このことにより，レジスタ干渉グラフのノードの次数を上げる可能性があり，レジスタ数の比較的少ない環境では不利になると思われる．

一方，後者のアプローチではコピー文の数は増えるが，その後のフェーズでこれらを都合良くコアレッシングすれば，前者のような状況は生じないと考えられていた．

このように，従来の SSA 逆変換アルゴリズムはそれぞれ異なった特徴を持つ．それにもかかわらず，これらの SSA 逆変換のアルゴリズムを実際に比較した研究はあまりなされていない．歴史的に遅れて登場した Sreedhar らの文献においても，他の SSA 逆変換アルゴリズムとの比較はされていない．このため，SSA 形式を最適化コンパイラに採用するにあたって，SSA 逆変換アルゴリズムに関して指標となるものがなかった．

そこで本研究では，SSA 形式のプログラム中間表現からの SSA 正規化について，Briggs らのアルゴリズムと Sreedhar らのアルゴリズムの長所と短所を明確にした．また，Briggs らのアルゴリズムにおいて，アドホックな処理についての改良案を提案した．さらに，Briggs らのアルゴリズムとその改良案を実装し，同一のコンパイラ上で Sreedhar らのアルゴリズムと比較した．とくに，複数のベンチマークでコアレッシングの影響やレジスタの数の多少との関係，複数の最適化との組合せを変えて評価し，結果に一般性を持たせた．

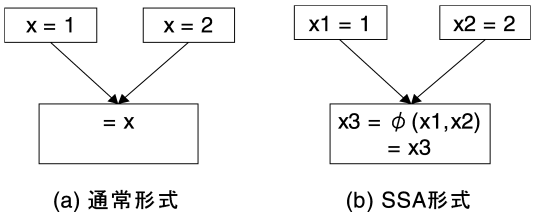

図 1 SSA 形式の例
Fig. 1 Example of SSA form.

本研究により，Briggs らの方法では，予想と異なり，コアレッシングすることのできないコピー文が数多く挿入されること，コアレッシングのできないコピー文は単にコピー文の実行に余分な時間がかかるだけでなく，レジスタを無駄に消費する可能性があること，最適化の組合せによって，逆変換アルゴリズムの違いによる影響はほとんど現れないこと，などが分かった．また，Sreedhar らの方法が実証的に最も優れていることが判明した．

2. SSA 形 式

SSA 形式はプログラムの中間表現の 1 つで，プログラム上の各変数の定義を字面上で 1 カ所になるように表現したものである[5]．また，異なる定義の合流点には **ϕ 関数**という仮想的な関数を挿入することで定義を 1 つにまとめる（**図 1**）．図 1 (b) での「x3 = ϕ (x1, x2)」は，制御の流れが左上の基本ブロックから来たときは x3 に x1 の値を代入し，右上の基本ブロックから来たときは x3 に x2 の値を代入する，という意味である．SSA 形式は，変数の定義と使用の関係が明確になるため，最適化に適した形であるといわれている．

しかし，SSA 形式で表されたプログラム中に含まれる ϕ 関数は一般的な計算機では実行不可能である．そこで実行可能なプログラムを得るためには，SSA 形式で表されたプログラムから ϕ 関数を削除し通常の形式（通常形式）に変換する必要がある．この変換を **SSA 逆変換**（SSA reverse translation）と呼ぶ．

3. SSA 逆変換

最初に，Cytron らの示した SSA 逆変換の基本的な方法[5] について述べる．**図 2** (a) の制御フローグラフを考えると，B1 から B3 へ制御が移ったとき，ϕ 関数

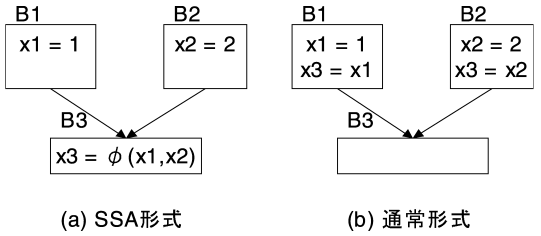

図 2 素朴な逆変換
Fig. 2 Naïve reverse translation.

によって x3 には x1 の値が代入される．そこで，B1 中の命令列の最後部に「x3 = x1」を挿入し，ϕ 関数を除去する（B2 のときも同様）．このように，ϕ 関数のあるブロックの先行ブロックに，適当なコピー文を挿入し，ϕ 関数を除去する方法を素朴な方法と呼ぶことにする．

素朴な方法ではプログラムの意味を変えてしまうような危うい例があることが知られている．素朴な逆変換を行うときに，そのような問題を引き起こす典型的な例として次の 2 つがある[1)]．

3.1 ロストコピー問題

1 つは，**図 3** (a) のようなプログラムを考える．このプログラムにコピー伝播を行うと B2 のコピー文「y = x1」が除去され，B3 の「return y」が「return x1」で置き換えられる（図 3 (b)）．

これを通常形式に逆変換することを考える．素朴な方法では，ϕ 関数のあるブロックの先行ブロックの最後尾に適当なコピー文を挿入する．この場合は，B1 に「x1 = x0」を，B2 に「x1 = x2」を挿入し，ϕ 関数を除去する（図 3 (c)）．しかし，図 3 (b) では x1 の生存区間が B2 の出口を含んでいる．ここで「x1 = x2」を B2 の最後尾に挿入してしまうと，x1 の生存区間内で不正に値を書き換えられることになってしまい，B3 において使用する x1 の値が破壊されてしまう．実際，図 3 (a)，(b) では B1-B2-B3 のように制御が流れた場合 B3 で返される値が 1 であるのに対して，図 3 (c) においては返される値が 2 となってしまい，プログラムの意味を変えてしまっている．これを**ロストコピー問題**と呼ぶ．これはクリティカルエッジが要因である．

このように，SSA 逆変換を行う際に挿入されるコピー文の左辺が挿入される場所で生存している場合は，注意が必要となる．

3.2 シンプルオーダリング問題

2 つ目は，同じブロック内に複数の ϕ 関数がある場合，それらの ϕ 関数は同時に処理されるものとして見なされなければならない．これを **ϕ 関数の同時代入性**と呼ぶことにする．

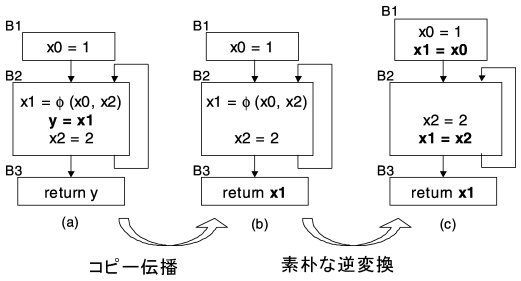

図 3 ロストコピー問題
Fig. 3 Lost copy problem.

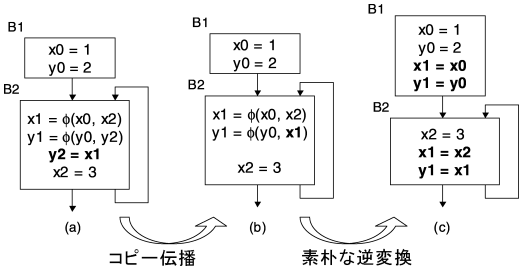

図 4 シンプルオーダリング問題
Fig. 4 Simple ordering problem.

今，**図 4** (a) のプログラムについて考える．このプログラムに対してコピー伝播が行われると B2 の「y2 = x1」が除去され，「y1 = ϕ(y0, y2)」が「y1 = ϕ(y0, x1)」に置き換えられる（図 4 (b)）．この場合のように同一ブロックに現れた 2 つの ϕ 関数は同時に処理されると考える．ここで，B1-B2-B2 のように制御が流れた場合，2 回目の B2 において y1 に代入される x1 の値は，2 回目の B2 で x1 に代入される値ではなく，1 回目の B2 で代入された値である．

ここで，図 4 (b) を逆変換することを考える．従来の方法では，

x1 = x2
y1 = x1

もしくは，

y1 = x1
x1 = x2

を B2 の最後尾へ挿入する．

ϕ 関数は同時代入性を持っているものとして処理されるべきものなので，本来なら挿入されるコピーの順序は関係ないはずである．しかし，B2 へ挿入されるコピー文の組を見ると前者と後者で y1 に入る値は異なってしまう（図 4 (c)）．これを**シンプルオーダリング問題**と呼ぶ．類似の問題としてスワップ問題と呼ばれるものもある[1)]．

このように同一ブロックに複数の ϕ 関数が存在する場合，SSA 逆変換する際に ϕ 関数の同時代入性に

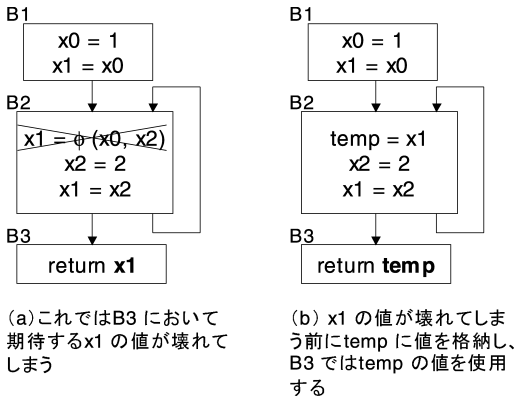

図 5 Briggs らの方法
Fig. 5 Method of Briggs, et al.

注意しなければならない.

4. Briggs らによる SSA 逆変換

Briggs らの逆変換アルゴリズム[1] は素朴な方法を拡張して，安全な逆変換を行う.

たとえば，ロストコピー問題の図 3 (b) に対し**図 5** (a) のように素朴な方法で逆変換を行った場合，B3 で使用している x1 は，挿入されたコピー文「x1 = x2」によって期待する値を壊されてしまう．そこで Briggs らの方法では，図 5 右のように，B2 の先頭に「temp = x1」を挿入して，x1 の値を temp に格納し，B2 以降で用いられている x1 は temp で置き換えるということをする．このように,「temp = x1」のようなコピー文を挿入することで素朴な方法の危うさを補い，プログラムの意味を変えずに逆変換を行うことができる.

上記のように Briggs らの逆変換アルゴリズムでは，素朴な逆変換で挿入されるコピー文を含め，多くのコピー文が挿入される．しかし彼らはコアレッシング（生存区間の合併，もとはレジスタ割当てで用いられた[3]）を行うことでその大部分を除去できると主張している．なお，我々の実装でも SSA 逆変換の後にコアレッシングを行っている.

5. Sreedhar らによる SSA 逆変換

Sreedhar らの逆変換アルゴリズム[19] では，素朴な方法や Briggs らの方法とはまったく異なったアプローチをとる．後者の 2 つのアルゴリズムは，ϕ 関数と同等の役割をするコピー文を挿入して ϕ 関数を除去する．それに対し，Sreedhar らのアルゴリズムでは ϕ 関数内の変数間に生存区間の干渉があるかどうかを調べ，干渉がなければそれらの変数を同じ変数で置き換えることで ϕ 関数を除去する.

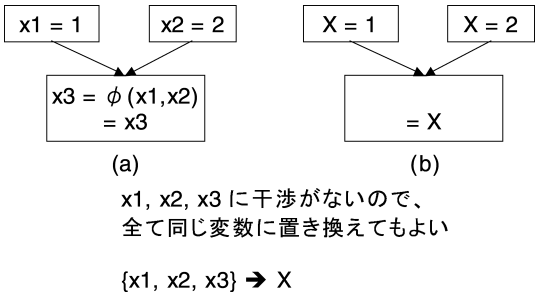

図 6 Sreedhar らの方法による ϕ 関数の除去
Fig. 6 Removal of ϕ-function by method of Sreedhar et al.

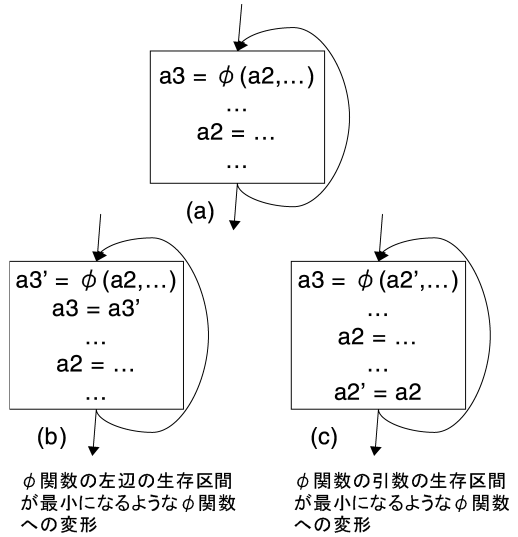

図 7 Sreedhar らの方法による ϕ 関数の書き換え
Fig. 7 Rewriting of ϕ-function by method of Sreedhar, et al.

たとえば，**図 6** (a) を考える．ここで，ϕ 関数内の変数（左辺を含む）である x1，x2，x3 の間に生存区間の干渉はない．そこで，x1，x2，x3 をすべて同じ変数 X に置き換える（本論文ではこれを同一化と呼ぶ）．すると，図 6 (b) のように，コピー文を挿入しなくても，ϕ 関数を除去し，通常形式に変換することができる.

しかし ϕ 関数内の変数の間に生存区間の干渉があった場合，合併を行うことができないため，これをなくす必要がある．そこで Sreedhar らの方法では，ϕ 関数を書き換えることで干渉を取り除くことをする．以下の作業を組み合わせて行い，ϕ 関数内の変数の生存区間を短くすることで干渉を避けることができる．**図 7** (a) が与えられたとする.

- ϕ 関数の左辺は，その関数が定義されているブロックの入口で生きていると考える．そこで ϕ 関数の左辺が最小の生存区間を持つようにするには,

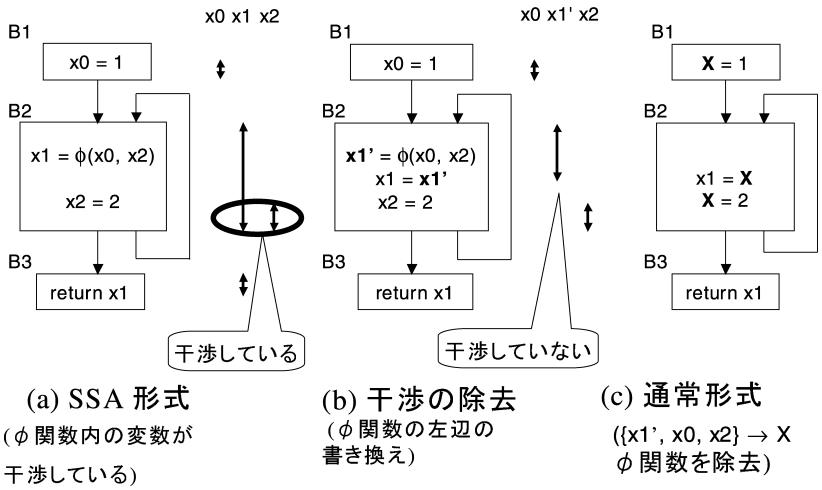

図 8 Sreedhar らの方法による ϕ 関数の書き換えの具体的な例
Fig. 8 Concrete example of rewriting of ϕ-function by method of Sreedhar, et al.

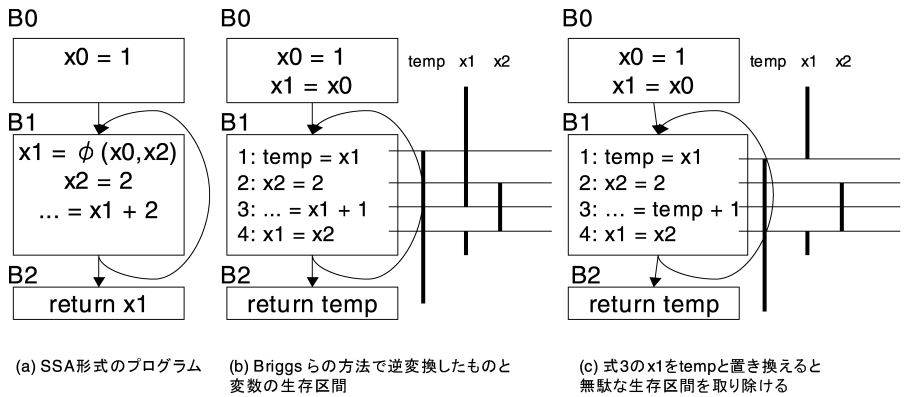

図 9 無駄な生存区間ができてしまう例
Fig. 9 Example of redundant live ranges.

図 7 (b) のように書き換えればよい.

- ϕ 関数の引数は，その対応する先行ブロックの出口で生きていると考える．そこで，引数が最小の生存区間を持つようにするには図 7 (c) のように書き換えればよい.

なお，Sreedhar らの方法では，逆変換した直後にさらにコアレッシングを行っている．

具体的な例について説明する．ロストコピー問題の**図 8** (a) を考えると，ϕ 関数内に現れる変数は x0，x1，x2 である．x1 と x2 の生存区間を見ると分かるように x1 と x2 の生存区間は干渉している．

この干渉をなくすには，ϕ 関数「x1 = ϕ(x0, x2)」の左辺である x1 の生存区間が B2 の出口を含まないようにしてやればよいので，ϕ 関数の左辺を書き換えてコピー文の挿入をする（図 8 (b)）．すると，新しい ϕ 関数内に現れる変数 x0，x1'，x2 は互いに干渉するものがなくなるので，x0，x1'，x2 をすべて同じ変数 X に置き換え，ϕ 関数を除去することができる（図 8 (c)）.

このように Sreedhar らの方法は，Briggs らの方法と比べて挿入されるコピー文が少ない.

なお Sreedhar らの方法では，逆変換時に SSA 形式に基づくコアレッシングも行っている.

6. 問題の提起と改良案

6.1 Briggs らの方法における欠点と改良案

6.1.1 無駄な生存区間によるもの

Briggs らの方法では，ϕ 関数の左辺の値を保持するために「temp = ϕ 関数の左辺」というコードを挿入することがある．このとき，ϕ 関数の左辺が temp と同じ値を持っているのに生存区間が重なってしまう場合があり，これは無駄な生存区間となる．たとえば，

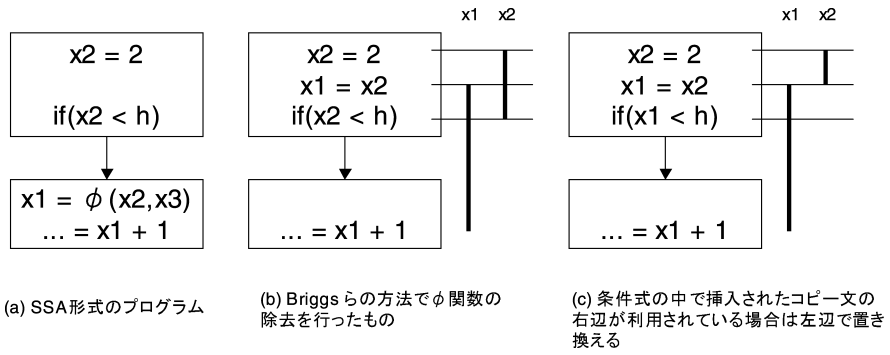

図 10 条件式によってできる生存区間の重なり
Fig. 10 Overlap of live ranges by conditional expression.

図 9 のような例を考える．

図 9 (b) において，x1 の値が temp に入っているにもかかわらず式 (3) で x1 を使用しているため x1 の生存区間は式 (3) までとなってしまっている．また，このような無駄な生存区間があるために x1 は x2 と干渉してしまっている．

この場合，式 (3) の x1 を temp で置き換える改良を行った．結果は図 9 (c) のようになる．このようにすると，式 (4) のコピー文もコアレッシング可能となる．

このような状況は必ずループの内側で生じるため，実行効率に与える影響が大きいと考える．

6.1.2 条件式によるもの

ブロックの最後が条件分岐命令のとき，ブロックの最後尾に挿入されるコピー文はその条件分岐命令の直前に挿入されることになる．このとき，挿入されたコピー文の右辺が条件式内で使用されているとコピー文の左辺と右辺の生存区間が重なってしまう．たとえば，**図 10** のような例を考える．

図 10 (b) において，x1 と x2 の生存区間に重なりが生じている．これは x2 と同じ値が x1 に入っているにもかかわらず，条件式で x2 を使用していることによるものである．このような場合，図 10 (c) のように条件式を書き換える改良を行った．これにより，「x1 = x2」はコアレッシング可能なものとなる．

6.2 Sreedhar らの方法における欠点

Sreedhar らの方法では ϕ 関数内の変数をすべて同じ変数に同一化することで ϕ 関数を除去する．ϕ 関数をコピー文と見なすと，レジスタアロケーションにおけるコアレッシングと同様の議論ができる．つまり，合併した変数の生存区間が長くなりレジスタ数の少ない環境では不利になる可能性がある．

この問題に対しては，7 章と 8 章の実験で検証を行う．

表 1 Sun-Blade-1000 の主な仕様
Table 1 The main specification of Sun-Blade-1000.

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

表 2 最適化の組合せ
Table 2 Combination of optimizations.

最適化 1	コピー伝播
最適化 2	コピー伝播，無用命令の除去，共通部分式除去
最適化 3	コピー伝播，ループ不変式の巻き上げ

7. 実 験 1

7.1 目的と評価の基準

本実験の目的は逆変換アルゴリズムの違いが実行効率にどのような影響を与えるかを調べ，逆変換アルゴリズムの選択指標とすることである．結果に一般性を持たせるため，レジスタの数，最適化の組合せを変えて評価を行う．

用いたコンパイラは COINS の C コンパイラ[4),17)]である．このコンパイラは「フロントエンド → SSA 変換 → 最適化 → SSA 逆変換 → Iterated coalescing によるレジスタ割当て[8)] → コード生成」という処理を行っている．また，COINS では命令スケジューリングがまだテスト段階であり，本実験で使用するいくつかのベンチマークでは正確に動作しないので，本実験では命令スケジューリングは行わない．

実験は SUN の Sun-Blade-1000 を用いて行った．主な仕様は**表 1** のとおりである．

レジスタの数は 20 本と 8 本で行う．20 本は SPARC や MIPS アーキテクチャのモデル化，8 本はレジスタ数の少ない組み込み用 CPU などでの傾向を調べるた

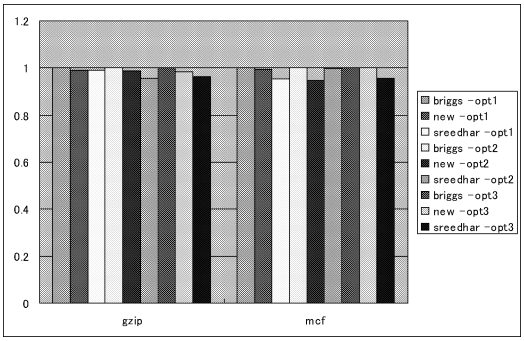

図 11 レジスタ 8 本での実行時間の相対比
Fig. 11 Ratio of execution time in 8 registers.

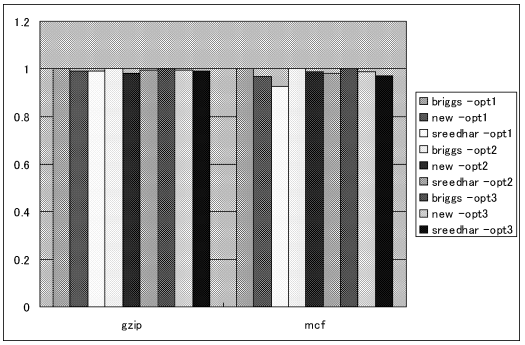

図 12 レジスタ 20 本での実行時間の相対比
Fig. 12 Ratio of execution time in 20 registers.

めにレジスタ数を制限したものである．最適化の組合せは，**表 2** のとおりである．なお，この章の図中のグラフでは，最適化 1 を「-opt1」，最適化 2 を「-opt2」，最適化 3 を「-opt3」と表記する．

ベンチマークには SPECint2000 から gzip，mcf を用いた．

7.2 実行時間による比較

実行時間による比較について述べる．

7.2.1 レジスタ 8 本の場合

図 11 にレジスタ数を 8 本に制限したときの実行時間の相対比（Briggs らの方法を 1 とした）を示す．new は 6.1 節で提案した Briggs らの方法の改良案である．

図から分かるように，Sreedhar らの方法は Briggs らの方法と比べて，gzip では最適化 1 を除いて 3〜4%ほど，mcf では最適化 2 を除いて 4〜5%ほど速い．

また，本研究で提案した改良案については，Briggs らの方法と比べて，gzip で 1%，mcf でも最適化 3 を除いてある程度の性能向上が見られる．

7.2.2 レジスタ 20 本の場合

次にレジスタ 20 本の場合の実行時間の相対比を**図 12** に示す．図から分かるように，Sreedhar らの方法は Briggs らの方法と比べて，gzip で 1%ほど，mcf では 2〜7%ほど速い．

本研究で提案した改良案については，Briggs らの方法と比べて，gzip で 1%，mcf でも 1〜3%ほどの性能向上が見られる．

以上の結果から，Sreedhar らの方法は Briggs らの方法と比べて実行効率に関して有利であることが分かった．これらに関する詳しい解析は文献 14) にある．しかし，改良案は若干の効果があったものの，Sreedhar らの方法の優位性を覆すほどのものではなかったため，次の章では Briggs らの方法と Sreedhar らの方法に絞ってより詳しくより多くのベンチマークを使用して評価する．

表 3 最適化の組合せ
Table 3 Combination of optimizations.

最適化 1	コピー伝播，定数伝播，無用命令除去，空ブロック除去
最適化 2	危険辺の除去，ループ不変式移動，定数伝播，共通部分式除去，コピー伝播，定数伝播，無用命令除去，空ブロック除去

8. 実 験 2

8.1 目的と評価の基準

本実験の目的は，Briggs らの方法と Sreedhar らの方法に絞って，前章より詳しくより多くのベンチマークで評価することである．

レジスタの数は前章と同様 8 本と 20 本で行う．最適化の組合せは**表 3** のとおりである．なお，この章の図中のグラフでは，最適化なしを「-opt0」，最適化 1 を「-opt1」，最適化 2 を「-opt2」と表記する．

ここまでの考察により Briggs らの方法では，(レジスタ割当てにおいて) コアレッシングできないコピー文が多く挿入されることが予想される (6.1 節)．コアレッシングできないコピー文はレジスタを無駄に消費する可能性があると考えられる．一方，Sreedhar らの方法では，レジスタ割当てでの積極的なコアレッシングと同様の問題点があることも予想される (6.2 節)．

そこでまず，静的な評価として，目的コード中のムーブ命令の数（コピー文の数），レジスタ割当て時のスピルの数を測定する．これは，コアレッシングできないコピー文が目的コードやレジスタ割当てに与える影響を確かめるためのものである．次に，動的な評価として，ムーブ命令の実行数，ロード・ストア命令の実行数を測定する．これは，静的に測定したものが実際に実行時に与える影響を確かめるためのものである．最後に，最終的な評価基準として実行時間を測定する．

ベンチマークには SPECint2000 から gzip，vpr，mcf，parser，bzip2，twolf を用いた．実験に用いた環境は実験 1 と同じである．

8.2 目的コード中のムーブ命令の数の比較

逆変換アルゴリズムの違いによって生じる，コアレッシングできないコピー文の数の差や，ϕ 関数内の変数を同一化処理することが目的コードにどのような影響を与えるかを調べるため，目的コード中のムーブ命令の数を測定する．ただし，この数はムーブ命令の動的な実行数を表したものではないため，この段階で優位性については判断できない．

8.2.1 レジスタ数 8 本の場合

図 13 に，レジスタ数を 8 本に制限したときの目的コード中のムーブ命令の数の相対比を示す（Briggs らの方法の最適化なしを 1 とする）．

図から分かるように，Sreedhar らの方法は Briggs らの方法と比べてムーブ命令の数が，60〜90%ほどだった．これによって，Briggs らの方法ではコアレッシングできないコピー文が多く挿入されることが確認できた．よって，Briggs らの方法では，実行時のムーブ命令の実行数も大きくなることが予想できる．これに関しては後の実験で確かめる．

最適化の組合せによる比較では，Briggs らの方法では最適化によってムーブ命令の数が少なくなっているのに対して，Sreedhar らの方法ではほとんど変わらなかった．これは，Sreedhar らの方法で行う同一化処理がある種の最適化と似た効果を持つためだと考えられる．

8.2.2 レジスタ数 20 本の場合

図 14 に，レジスタ数を 20 本にして同様の実験を行った結果を示す．

図から分かるように，Sreedhar らの方法は Briggs らの方法と比べてムーブ命令の数が，50〜80%ほどだった．これは，レジスタ数 8 本のときより大きな差となった．

最適化の組合せによる比較では，レジスタ数 8 本のときと同様に，Briggs らの方法では最適化によってムーブ命令の数が少なくなっているのに対して，Sreedhar らの方法ではほとんど変わらなかった．これに関してはレジスタ数 8 本のときと同様の議論ができる．

8.3 スピル数の比較

SSA 逆変換アルゴリズムの違いによって生じる，コアレッシングできないコピー文の数の差や，ϕ 関数内の変数を同一化処理することがレジスタ割当てに与える影響を調べるため，レジスタ割当て時にスピルされる変数の数を調べる．

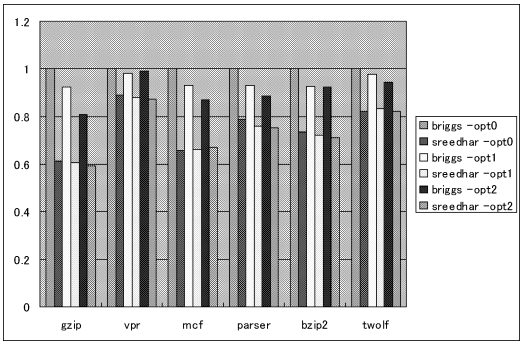

図 13 レジスタ 8 本でのムーブ命令数の相対比
Fig. 13 Ratio of number of moves in 8 registers.

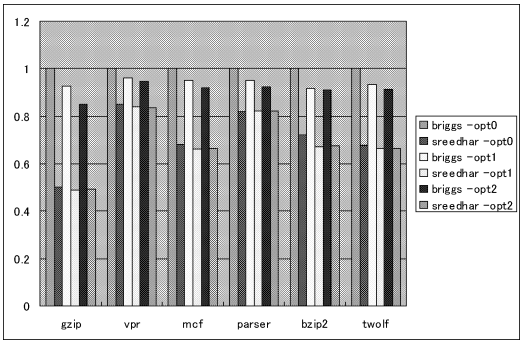

図 14 レジスタ 20 本でのムーブ命令数の相対比
Fig. 14 Ratio of number of moves in 20 registers.

ただし，ムーブ命令の比較と同様に，この数はスピルされることによる動的なコストを表したものではないため，この段階で優位性については判断できない．

8.3.1 レジスタ数 8 本の場合

図 15 に，レジスタ数を 8 本に制限したときのレジスタ割当て時のスピルの数を示す．

図から分かるように，Sreedhar らの方法は Briggs らの方法と比べてスピルされる変数の数が少ない．これは，Briggs らの方法ではコアレッシングできないコピー文によってレジスタを無駄に消費しているためだと思われる．これによって実行時のロード・ストア命令の実行数が大きくなることが予想できる．これに関しても，後の実験で確かめる．

8.3.2 レジスタ数 20 本の場合

図 16 に，レジスタ数を 20 本にしたときのレジスタ割当て時のスピルの数を示す．

図から分かるように，レジスタ数 8 本のときと同様に，Sreedhar らの方法は Briggs らの方法と比べてスピルされる変数の数が少ない．また，レジスタ数 8 本のときよりスピルの数は少ない．これによって，レジスタ数 8 本のときよりロード・ストア命令の実行数に与える影響は小さいと予想できる．これに関しても，

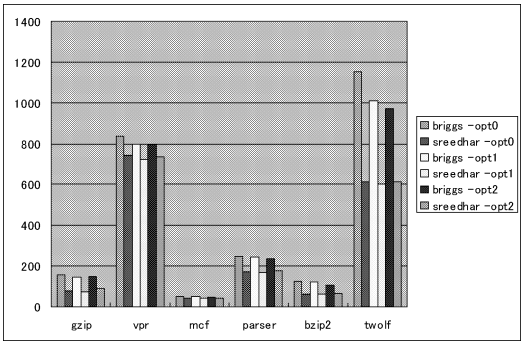

図 15 レジスタ 8 本でのスピル数
Fig. 15 Number of spills in 8 registers.

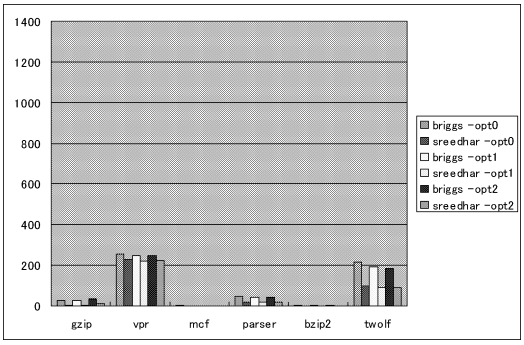

図 16 レジスタ 20 本でのスピル数
Fig. 16 Number of spills in 20 registers.

後の実験で確かめる．

8.4 ムーブ命令の実行数による比較

静的な測定では Sreedhar らの方法の方が Briggs らの方法に比べてムーブ命令の数が少なかった．ここでは，実行時のムーブ命令の実行数を測定する．

8.4.1 レジスタ 8 本の場合

図 17 に，レジスタ数を 8 本に制限したときのムーブ命令の実行数の相対比を示す．

図から分かるように，Sreedhar らの方法は Briggs らの方法と比べて，gzip では 10%，その他では 50〜80%ほどの実行数であった．

最適化の組合せによる比較では，静的な測定と類似の傾向があり，Sreedhar らの方法では最適化によってムーブ命令の実行数はそれほど変わらなかった．

8.4.2 レジスタ 20 本の場合

図 18 に，レジスタ数を 20 本にしたときのムーブ命令の実行数の相対比を示す．

図から分かるように，Sreedhar らの方法は Briggs らの方法と比べて，gzip では 3%，その他では 40〜80%ほどの実行数であった．これも，静的な測定と同様にレジスタ 8 本のときと比べて差が大きくなっている．

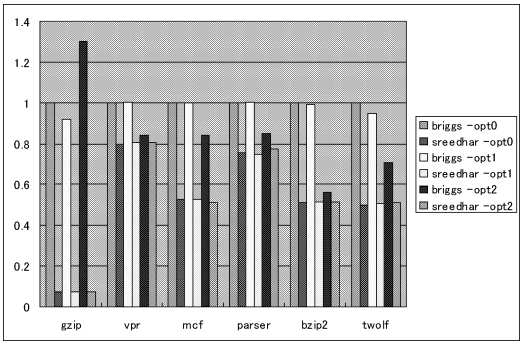

図 17 レジスタ 8 本でのムーブ命令の実行数の相対比
Fig. 17 Ratio of number of execution of moves in 8 registers.

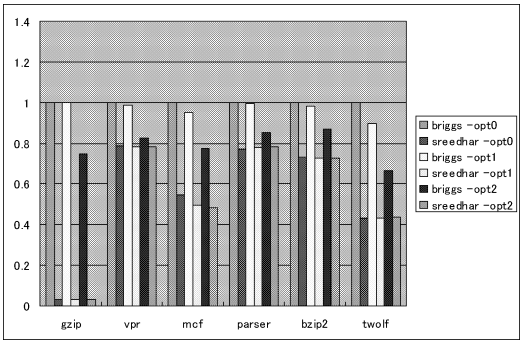

図 18 レジスタ 20 本でのムーブ命令の実行数の相対比
Fig. 18 Ratio of number of execution of moves in 20 registers.

最適化の組合せによる比較では，レジスタ数 8 本のときにもまして，Sreedhar らの方法では最適化によってムーブ命令の実行数がほとんど変わらなかった．

8.5 ロード・ストア命令の実行数による比較

静的な測定では，Sreedhar らの方法の方がスピルの数が少なかった．

しかし，Sreedhar らの方法は ϕ 関数内の変数を同一化しているので，ϕ 関数内の変数の生存区間が長くなり，変数 1 つあたりのスピルによるコストが大きいと考えられる．

そこで，実行時にかかるスピルのコストの近似として，ロード・ストア命令の実行数を測定した．

8.5.1 レジスタ数 8 本の場合

図 19 に，レジスタ数を 8 本に制限したときのロード・ストア命令の実行数の相対比を示す．

図から分かるように，ロード・ストア命令の実行数において，ベンチマーク 6 つ中 4 つが Sreedhar 有利，2 つが Briggs 有利という結果になった．とくに gzip においては Sreedhar が大きく有利だった．

また，Sreedhar らの方法はスピルされる変数の数

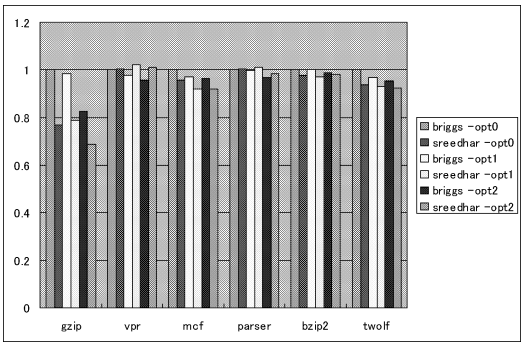

図 19 レジスタ 8 本でのロード・ストアの実行数の相対比
Fig. 19 Ratio of number of execution of load & store in 8 registers.

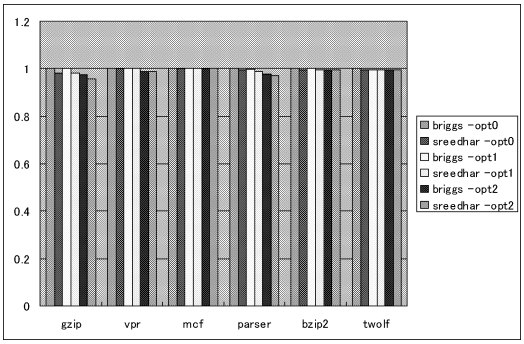

図 20 レジスタ 20 本でのロード・ストアの実行数の相対比
Fig. 20 Ratio of number of execution of load & store in 20 registers.

が圧倒的に少ない割に，ロード，ストア命令の実行数ではその差が縮まっており，一部では逆転していることから，スピルされる変数 1 つあたりのコストが高いことも分かる．

最適化の組合せを変えても，2 つの方法での差は保たれている．

8.5.2 レジスタ数 20 本の場合

図 20 に，レジスタ数を 20 本にしたときのロード・ストア命令の実行数の相対比を示す．

図から分かるように，レジスタ 8 本のときほどの差は現れなかった．これは図 16 においてスピルされた変数の数の差が図 15 よりも少なかったことによるものであると考えられる．

このように，ロード・ストア命令の実行数にほとんど差がないことから，レジスタ数を 20 本にしたときの実行時間については，ムーブ命令の実行数の差に依存してくるのではないかと予測される．

8.6 実行時間による比較

次に実行時間による比較について述べる．

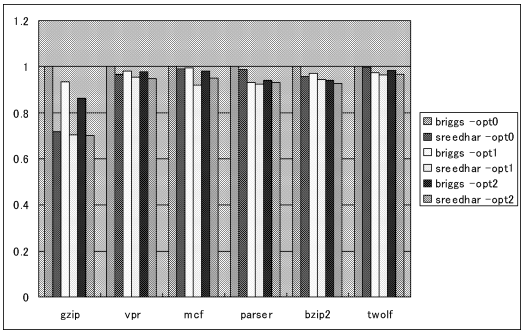

図 21 レジスタ 8 本での実行時間の相対比
Fig. 21 Ratio of execution time in 8 registers.

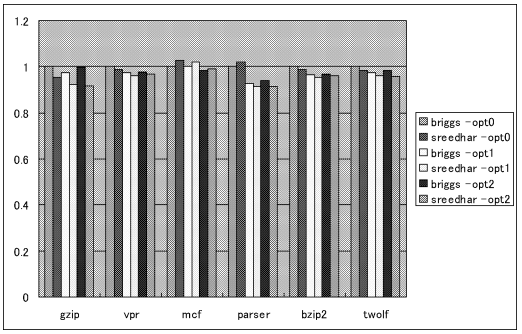

図 22 レジスタ 20 本での実行時間の相対比
Fig. 22 Ratio of execution time in 20 registers.

8.6.1 レジスタ数 8 本の場合

レジスタ数を 8 本にしたときの実行時間の相対比を**図 21** に示す．相対比から分かるように，Sreedhar らの方法は Briggs らの方法と比べて，gzip では 28%ほど，その他でもすべて若干速い．実行時間とその相対比のばらつきについては後でもう少し詳しく考察する．

8.6.2 レジスタ数 20 本の場合

次にレジスタ数を 20 本にしたときの実行時間の相対比を**図 22** に示す．Sreedhar らの方法は Briggs らの方法と比べて，mcf のすべての最適化の組合せ，parser の最適化なしでは最大 3%ほど遅い．その他では最大 8%ほど速い．

実行時間の比較では，おおむねここまでの考察のとおりの結果であったが，一部でこれとは異なり，ばらつきが見られた．

このようなばらつきは SSA 正規化アルゴリズムによる影響よりも，アーキテクチャによる命令処理の並列性の影響によるものであると思われる．原因の 1 つとして考えられるのは，命令のフェッチサイクルの問題である．たとえば，ループの中で分岐命令があったとき，この分岐命令を処理している間に分岐予測に従って次の一連の命令をフェッチしようとする．このとき

フェッチしたい命令がキャッシュラインに綺麗に乗っていればまとめてフェッチすることができるが，キャッシュラインの境界をまたがっていた場合には余分に時間がかかる．実際，実行時間の大半を占めるループの命令をキャッシュラインに合わせてアライメントしなおすと，実行時間が4〜5%向上することがある[14]．このような遅延は今回実験に用いたようなスーパスカラマシンではとくに影響が大きいと考えられている．

このように，実行時間は，逆変換アルゴリズムの違いによる影響が他の要因によって隠蔽されることがあったが，平均的に見て，Sreedhar らの方法に優位性があると思われる．

以上をまとめると，レジスタ数 8 本の場合は Sreedhar らの方法が実行時間が短い．これは，スピルの動的なコスト，ムーブ命令の実行数の少なさによると考えられる．一方，レジスタ数 20 本の場合は，大勢として Sreedhar らの方法が実行時間が短い．これは，ムーブ命令の実行数の少なさが効いていると思われる．また，最適化の組合せを変えても，SSA 逆変換アルゴリズムの違いによる影響はほとんどない．

9. 命令スケジューリングについての議論

本実験では，7.1 節で述べたように，命令スケジューリングは行わなかった．しかし，最近の多くのプロセッサでは命令レベル並列処理を行っているので，命令スケジューリングは性能に大きな影響を与えると思われる．また，命令スケジューリングはレジスタ割当てと互いに干渉し合うことが知られている．すなわち，レジスタ割当てで実レジスタ数を減少させると命令間の並列度が上がらず，逆に命令スケジューリングで並列度を上げようとするとレジスタ圧力が増す．

これを本研究に即していえば，コアレッシングによってレジスタの再利用性を高めると，命令どうしの依存が増えて，逆に並列度が下がってしまうことがあり，out-of-order 実行を行うプロセッサでは，命令の入れ替えを十分に行えなくなる可能性もある．

レジスタ割当てと命令スケジューリングは互いに依存するため，適用の順番や協調動作について多くの研究がなされている[13]．

COINS コンパイラにおいても，命令スケジューリングの試験的実装として，「命令スケジューリング → レジスタ割当て → 命令スケジューリング」の順での適用が試みられ，小さい例題では効果が上がっている．

本研究では SSA 逆変換のアルゴリズムとして，Briggs らの方法と Sreedhar らの方法がレジスタ割当てに与える影響について評価をした．しかし，逆変換がレジスタ割当てに良い影響を与えても，命令スケジューリングに悪い影響を与える可能性も考えられるので，今後，命令スケジューリングを行った状態での評価が必要であると考えられる．

また，レジスタリネーミングをハードウェアで実行するプロセッサも存在するので，そのようなプロセッサではまた別の視点での評価が必要であろう．

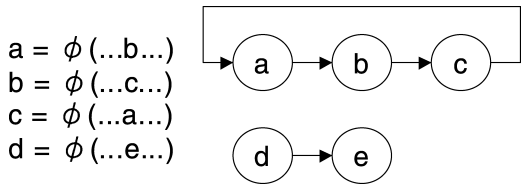

図 23 Morgan の方法
Fig. 23 Method of Morgan.

10. 関 連 研 究

本研究は SSA 逆変換アルゴリズムを比較することを目的としているが，このような研究は他になされていない．そこで，本章ではその他の SSA 逆変換アルゴリズムについて述べる．

10.1 Morgan による SSA 逆変換

まず Morgan の逆変換[16] について簡単に説明する．このアルゴリズムは Briggs らと同様に ϕ 関数をコピー文に置き換えるものである．Morgan の逆変換では**図 23** のようなグラフを用いる．このグラフは，ノードを変数とし，ϕ 関数の左辺から引数へ有向辺を引いたものである．このグラフにおいて，先行ノードを持たない変数（ノード）は，ϕ 関数の引数ではない．つまり，その変数は逆変換によって書き換えられてもよいので，安全にコピー文「自分 = 後続ノード」を挿入できる．また，このグラフがサイクルを持つとき，挿入するべきコピー文の間に依存関係ができてしまう．そこで，一時変数を使い問題を回避する．たとえば，図 23 において，d は先行ノードを持たないので，「d = e」を挿入することができる．次に，a→b→c→a はサイクルになっているので，まず「temp = a」を挿入して a の値を保持し，有向辺に沿ってコピー文を挿入すればよい．つまり，コピー文の順序は次のようになる．

d = e
temp = a
a = b
b = c
c = temp

Briggs らの方法でもこのような場合を扱え，またあ

る形のサイクルについては Briggs らの方法の方がコピー文が少なくてすむので，この方法は Briggs らの方法のサブセットになっている．

10.2 Budimlic らによる SSA 逆変換

次に Budimlic らの逆変換[2] について簡単に説明する．この逆変換アルゴリズムは主に JIT コンパイラ向けに開発されたもので，本研究で対象としたものとは異なり，コンパイル時間の軽減を目指すものである．基本的なアイディアは Sreedhar らと同様，ϕ 関数を同一化することで逆変換を行う．このアルゴリズムでは，まず SSA 形式で表されたプログラムの性質を用いて生存区間を近似する．この近似された生存区間を用いて ϕ 関数の干渉を取り除くことと，干渉の場合分けが Sreedhar らの方法よりも簡易であることから，挿入されるコピー文の数は Sreedhar らの方法より多くなると思われる．

11. おわりに

SSA 形式からの逆変換アルゴリズムには，主要な 2 つのアプローチが存在するものの，これまでそれらを比較する研究はなされてこなかった．そのため，初期のアルゴリズムの誤りを最初に解決した Briggs らの方法をそのまま採用するケースが多かった．そこで本研究では，逆変換アルゴリズムの異なるアプローチに対して，長所と短所を明確にし，検証を行った．これにより，今まで注目されていなかった逆変換アルゴリズムがプログラムの実行効率に少なからず影響を与えることを明らかにし，今後，逆変換アルゴリズムを選択する際の指標を示したことが，本研究の最大の意義であると考える．

得られた知見として，

- Briggs らの方法ではコアレッシングできないコピー文が大量に挿入され，Sreedhar らの方法によって懸念されるレジスタ圧力の増加以上に影響を与えること，
- レジスタ数の比較的少ない環境では，スピルによる動的なコストと，コピー文の実行数という 2 つの理由から，ϕ 関数に対してコアレッシングに類似した処理をするアプローチである Sreedhar らの方法の方が，実行時間で数%，最大 28%ほど有利となること，
- レジスタ数の比較的多い環境でも，コピー文の実行数という理由から，Sreedhar らの方法が多くの場合に実行時間が数%ほど有利であること，

などがあげられる．

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) Briggs, P., Cooper, K.D., Harvey, T.J. and Simpson, L.T.: Practical Improvements to the Construction and Destruction of Static Single Assignment Form, *Software — Practice and Experience*, Vol.28, No.8, pp.859–881 (1998).
2) Budimlic, Z., Cooper, K.D., Harvey, T.J., Kennedy, K., Oberg, T.S. and Reeves, S.W.: Fast Copy Coalescing and Live-Range Identification, *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp.25–32 (2002).
3) Chaitin, G.J.: Register Allocation & Spilling via Graph Coloring, *Proc. SIGPLAN '82 Symposium on Compiler Construction*, pp.98–105 (1982).
4) COINS Project：並列化コンパイラ向け共通インフラストラクチャCOINS ホームページ．http://www.coins-project.org/
5) 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. Prog. Lang. Syst.*, Vol.13, No.4, pp.451–490 (1991).
6) Fitzgerald, R., Knoblock, T.B., Ruf, E., Steensgaard, B. and Tarditi, D.: Marmot: An Optimizing Compiler for Java, *Software — Practice and Experience*, Vol.30, No.3, pp.199–232 (2000).
7) GCC: GCC Homepage. http://gcc.gnu.org/
8) George, L. and Appel, A.W.: Iterated Register Coalescing, *ACM Trans. Prog. Lang. Syst.*, Vol.18, No.3, pp.300–324 (1996).
9) Harvard University: MachSUIF Homepage. http://www.eecs.harvard.edu/machsuif/
10) IBM: Jikes Research Virtual Machine. http://jikesrvm.sourceforge.net/
11) Intel: Intel Compilers. http://developer.intel.com/software/products/compilers
12) Ishizaki, K., Takeuchi, M., et al.: Effectiveness of Cross-platform Optimizations for a Java Just-in-time Compiler, *OOPSLA '03: Proc. 18th annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages and Applications*, pp.187–204, ACM Press (2003).
13) 片岡正樹，小関 聰，小松秀昭，深澤良彰：レジスタ生存グラフを用いたレジスタ割付けへのプロセッサ並列度の考慮，情報処理学会論文誌：

プログラミング，Vol.46, No.SIG 1 (PRO 24), pp.10–18 (2005).
14) 小濱真樹：SSA 正規化アルゴリズムの比較と評価，東京工業大学数理・計算科学専攻修士論文 (2004).
15) 丸山冬彦：JIT コンパイラ向けアプリケーションフレームワークの設計と実装，東京工業大学大学院情報理工学研究科数理・計算科学専攻修士論文 (2001).
16) Morgan, R.: *Building an Optimizing Compiler*, Digital Press (1998).
17) Sassa, M., Nakaya, H., Kohama, M., Fukuoka, T. and Takahashi, M.: Static Single Assignment Form in the COINS Compiler Infrastructure, *SSGRR 2003w International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, e-Medicine on the Internet* (2003).
18) Scale Compiler Group: University of Massachusetts. http://www-ali.cs.umass.edu/Scale/
19) Sreedhar, V.C., Ju, R.D.-C., Gillies, D.M. and Santhanam, V.: Translating Out of Static Single Assignment Form, *Proc. 6th International Symposium on Static Analysis*, Lecture Notes in Computer Science, Vol.1694, pp.194–210, Springer-Verlag (1999).

(平成 17 年 2 月 21 日受付)
(平成 17 年 7 月 13 日採録)

伊藤　陽

1981 年生．2004 年東京工業大学理学部情報科学科卒業．同年同大学大学院情報理工学研究科数理・計算科学専攻入学，現在に至る．

小濱　真樹

1979 年生．2002 年東京工業大学大学院情報理工学研究科数理・計算科学専攻入学．2004 年同大学院情報理工学研究科数理・計算科学専攻修了．同年富士写真フイルム（株）入社，現在に至る．

佐々　政孝（正会員）

1948 年生．1970 年東京大学理学部物理学科卒業．1974 年同大学大学院博士課程中退，東京工業大学理学部情報科学科助手．1981 年筑波大学電子・情報工学系．1992 年東京工業大学理学部．現在同大学大学院情報理工学研究科数理・計算科学専攻教授．理学博士．プログラミング言語，コンパイラ，プログラミング環境に興味を持つ．著書『プログラミング言語処理系』（岩波書店）．日本ソフトウェア科学会，ACM，IEEE 各会員．