

## 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<sup>†</sup> 小濱 真樹<sup>††</sup> 佐々 政 孝<sup>†</sup>

SSA 形式 (静的単一代入形式) は,  $\phi$  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一ヶ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  $\phi$  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,  $\phi$  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ス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, 改良案は若干ながら効果があること, 実証的には Sreedhar らの方法がもっとも優れていること, が判明した.

## Comparison and Evaluation of Back Translation Algorithms for Static Single Assignment Form

YO ITO,<sup>†</sup> MASAKI KOHAMA<sup>††</sup> and MASATAKA SASSA<sup>†</sup>

The SSA form (static single assignment form) is an intermediate representation where the definition of each variable appears only once in the text by using a virtual function called  $\phi$ -function. However, since the  $\phi$ -functions appearing in the program using the SSA form are virtual, the target code cannot be directly generated from the SSA form. Therefore, it is necessary to delete the  $\phi$ -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 second 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 a different characteristic feature. Therefore, there has been no criterion as to which SSA reverse translation algorithm should be selected when the SSA form is used in the optimizing compiler.

In this paper, we clarified merits and weak points of algorithms by Briggs and Sreedhar. We also proposed an improvement of the algorithm by Briggs. Then, we implemented algorithms by Briggs and its improvement and the algorithm by Sreedhar on the same compiler, and compared 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, and our improvement was effective to a certain extent. Empirically, the method by Sreedhar was the most efficient.

### 1. はじめに

SSA 形式 (静的単一代入形式) は,  $\phi$  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一ヶ所だけになるように表した中間表現である. SSA 形

<sup>†</sup> 東京工業大学  
Tokyo Institute of Technology  
<sup>††</sup> 富士写真フイルム  
Fuji Photo Film Co., Ltd.

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<sup>(6),(7),(9)~(11),(17)</sup>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 $\phi$ 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、 $\phi$ 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<sup>5)</sup>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事が知られている<sup>1)</sup>。

実用的なアルゴリズムとしてはBriggsらの提案した方法<sup>1)</sup>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<sup>18)</sup>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<sup>6)</sup>、Machine SUIF<sup>9)</sup>、Scale<sup>17)</sup>などがあり、Sreedharらの方法を用いたものにはOpenJIT<sup>12),(14)</sup>などがある。

Sreedharらの方法では $\phi$ 関数に対し、ある種のコアレスニング(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 $\phi$ 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度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スニングすれば、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。それにも関わらず、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

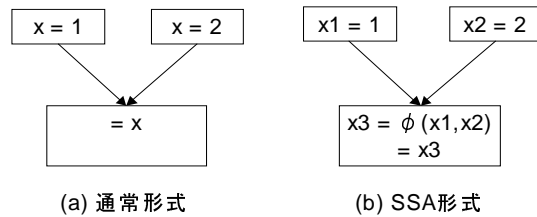


図1 SSA形式の例  
Fig. 1 Example of SSA Form

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スニ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み合わせを変えて評価し、結果に一般性を持たせた。

本研究により、Briggsらの方法では、予想と異なり、コアレスニ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、コアレスニ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、最適化の組み合わせによって、逆変換アルゴリズムの違いによる影響はほとんど現われないこと、などがわかった。また、Sreedharらの方法が実証的に最も優れていること、改良案は若干ながら効果があることなどが判明した。

## 2. SSA形式

SSA形式はプログラムの中間表現の一つで、プログラム上の各変数の定義を字面上で一ヶ所になるように表現したものである<sup>5)</sup>。また、異なる定義の合流点には $\phi$ 関数という仮想的な関数を挿入することで定義を一つにまとめる(図1)。図1(b)での「 $x3 = (x1, x2)$ 」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

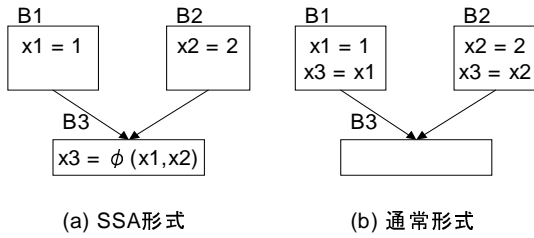


図 2 素朴な逆変換

Fig. 2 Naive reverse translation

る  $\phi$  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  $\phi$  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<sup>5)</sup> 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 へ制御が移った時， $\phi$  関数によって  $x_3$  には  $x_1$  の値が代入される．そこで，B1 中の命令列の最後部に「 $x_3 = x_1$ 」を挿入し， $\phi$  関数を除去する (B2 の時も同様)．このように， $\phi$  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， $\phi$  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要因として次の二つが考えられる<sup>1)</sup>．

#### 3.1 ロストコピー問題

一つは，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「 $y = x_1$ 」が除去され，B3 の「 $\text{return } y$ 」が「 $\text{return } x_1$ 」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，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「 $x_1 = x_0$ 」を，B2 に「 $x_1 = x_2$ 」を挿入し，関数を除去する (図 3(c))．しかし，図 3(b) では  $x_1$  の生存区間が B2 の出口を含んでいる．ここで「 $x_1 = x_2$ 」を B2 の最後尾に挿入してしまうと， $x_1$  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  $x_1$  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

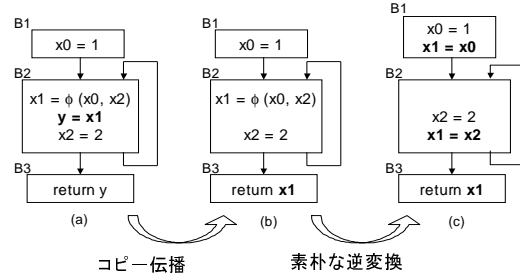


図 3 ロストコピー問題

Fig. 3 Lost copy problem

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 シンプルオーダーリング問題

二つ目は，同じブロック内に複数の関数がある場合，それらの関数は同時に処理されるものとしてみなされなければならない．これを関数の同時代入性と呼ぶことにする．

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「 $y_2 = x_1$ 」が除去され「 $y_1 = (y_0, y_2)$ 」が「 $y_1 = (y_0, x_1)$ 」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 つの関数は同時に処理され则认为．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  $y_1$  に代入される  $x_1$  の値は，2 回目の B2 で  $x_1$  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，

$$x_1 = x_2$$

$$y_1 = x_1$$

もしくは，

$$y_1 = x_1$$

$$x_1 = x_2$$

を B2 の最後尾へ挿入する．

関数は同時代入性をも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  $y_1$  に入る値は異なってしま (図 4(c))．これをシンプルオーダーリング問題と呼ぶ．類似の問題としてスワップ問題と呼ばれるものもある<sup>1)</sup>．

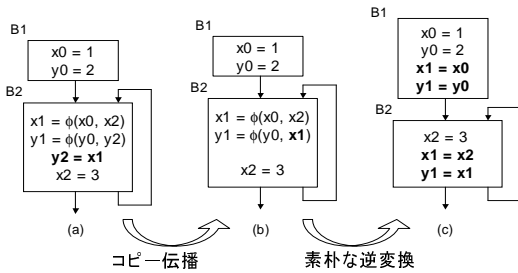


図 4 シンプルオーダーリング問題  
Fig. 4 Simple ordering problem

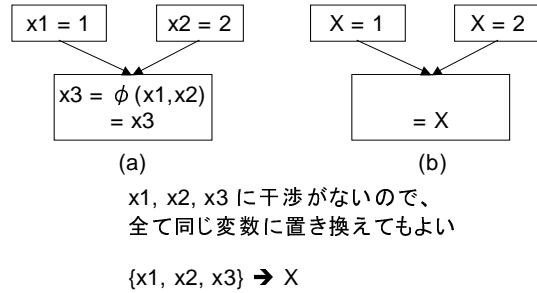
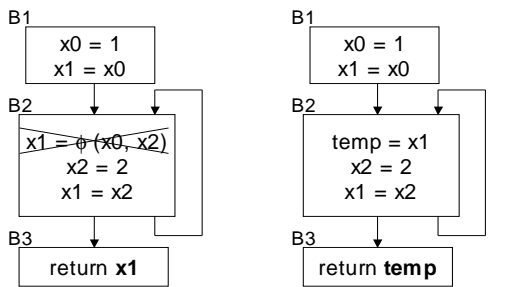


図 6 Sreedhar らの方法による  $\phi$  関数の除去  
Fig. 6 Removal of  $\phi$ -function by method of Sreedhar et al.



(a) ここでは B3 において期待する  $x1$  の値が壊れてしまう  
(b)  $x1$  の値が壊れてしまう前に temp に値を格納し、B3 では temp の値を使用する

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

このように同一ブロックに複数の関数が存在する場合、SSA 逆変換する際に関数の同時代入性に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<sup>1)</sup> は素朴な方法を拡張して、安全な逆変換を行う。

例えば、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スニング(生存区間の合併、もとはレジスタ割り当てで用いら

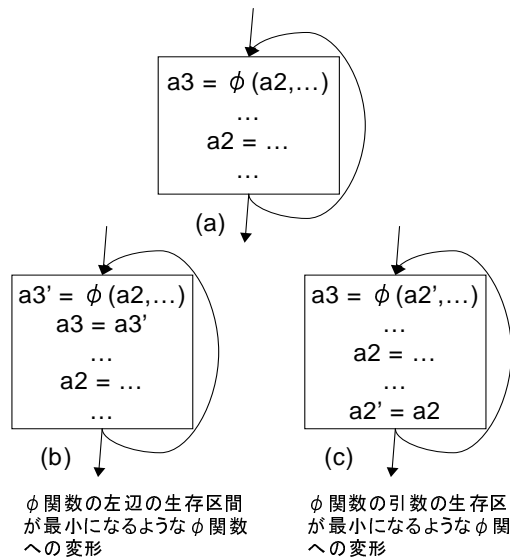


図 7 Sreedhar らの方法による  $\phi$  関数の書き換え  
Fig. 7 Rewriting of  $\phi$ -function by method of Sreedhar

れた<sup>3)</sup> ) 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スニングを行っている。

#### 5. Sreedhar らによる SSA 逆変換

Sreedhar らの逆変換アルゴリズム<sup>18)</sup> 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。後者の二つのアルゴリズムは、 $\phi$  関数と同等の役割をするコピー文を挿入して  $\phi$  関数を除去する。それに対し、Sreedhar らのアルゴリズムでは  $\phi$  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  $\phi$  関数を除去する。

例えば、図 6(a) を考える。ここで、 $\phi$  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、 $x1, x2, x3$  を全て同じ変数  $X$  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、 $\phi$  関数を除去し、通常形式に変換することができる。

しかし  $\phi$  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、 $\phi$  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、 $\phi$  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。

- $\phi$  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 関数の左辺が最小の生存区間を持つようにするには、図 7(b) のように書き換えればよい。
- $\phi$  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、関数内に現れる変数は  $x_0, x_1, x_2$  である。 $x_1$  と  $x_2$  の生存区間を見るとわかるように  $x_1$  と  $x_2$  の生存区間は干渉している。

この干渉を無くすには、関数「 $x_1 = (x_0, x_2)$ 」の左辺である  $x_1$  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、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、新しい関数内に現れる変数  $x_0, x_1', x_2$  は互いに干渉するものがなくなるので、 $x_0, x_1', x_2$  を全て同じ変数  $X$  に置き換え、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、 $\phi$  関数の左辺の値を保持するために「temp = 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、 $\phi$  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。例えば、図 9 のような例を考える。

図 9(b) において、 $x_1$  の値が temp に入っているに

も関わらず式 3 で  $x_1$  を使用しているため  $x_1$  の生存区間は式 3 までとなってしまう。また、このような無駄な生存区間があるために  $x_1$  は  $x_2$  と干渉してしまっている。

この場合、式 3 の  $x_1$  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時、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。例えば、図 10 のような例を考える。

図 10(b) において、 $x_1$  と  $x_2$  の生存区間に重なりが生じている。これは  $x_2$  と同じ値が  $x_1$  に入っているにも関わらず、条件式で  $x_2$  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、「 $x_1 = x_2$ 」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  $\phi$  関数内の変数を全て同じ変数に同一化することで  $\phi$  関数を除去する。 $\phi$  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。

## 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、最適化の組み合わせを変えて評価を行なう。

用いたコンパイラは COINS の C コンパイラ<sup>4),16)</sup>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 によるレジスタ割り当て<sup>8)</sup> → コード生成」という処理を行っている。実験は SUN の Sun-Blade-1000 を用いて行った。主な仕様は表 2 の通りである。

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 x86 などのモデル化である。最適化の組み合わせは、表 1 の

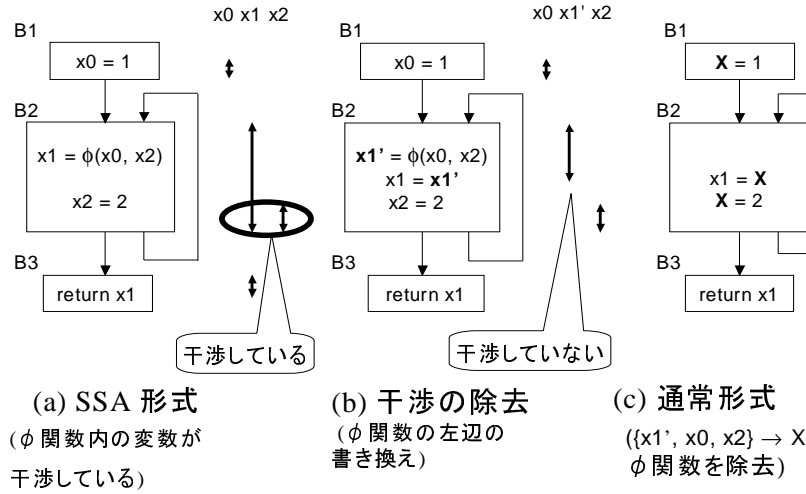


図 8 Sreedhar らの方法による  $\phi$  関数の書き換えの具体的な例  
Fig. 8 Concrete example of rewriting of  $\phi$ -function by method of Sreedhar

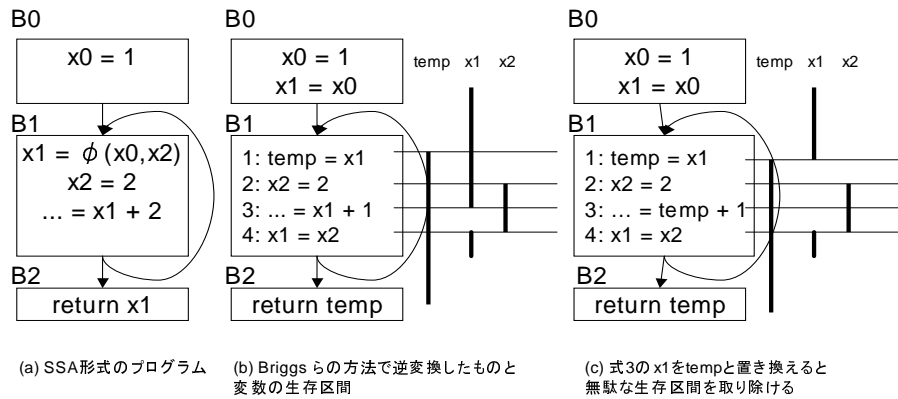


図 9 無駄な生存区間ができてしまう例  
Fig. 9 Example of redundant live ranges

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

表 1 最適化の組み合わせ  
Table 1 Combination of optimizations

通りである。なお、この節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 本に制限した時の実行時間の相対比 (Briggs らの方法を 1 とした) を示す。new

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 2 Sun-Blade-1000 の主な仕様  
Table 2 The main specification of Sun-Blade-1000

は 6.1 で提案した Briggs らの方法の改良案である。

図から判るように、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%程、mcf では最適化 2 を除いて 4~5%程速い。

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。

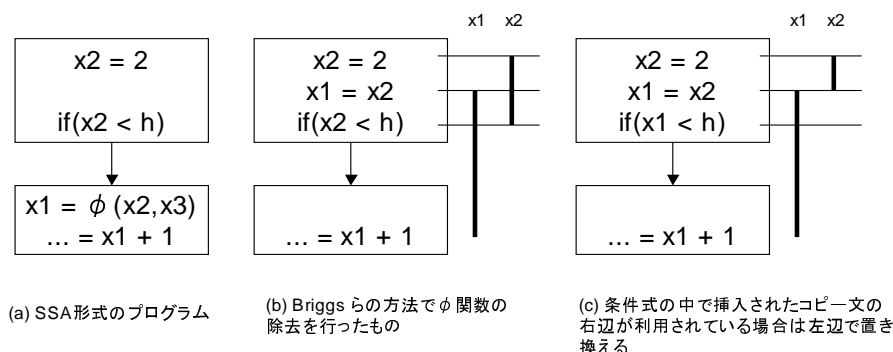


図 10 条件式によってできる生存区間の重なり

Fig. 10 Overlap of live ranges by conditional expression

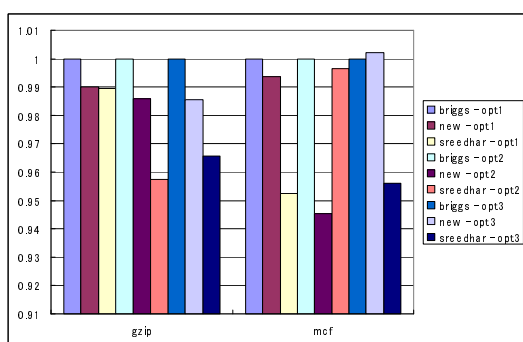


図 11 レジスタ 8 本での実行時間の相対比

Fig. 11 Ratio of execution time in 8 registers

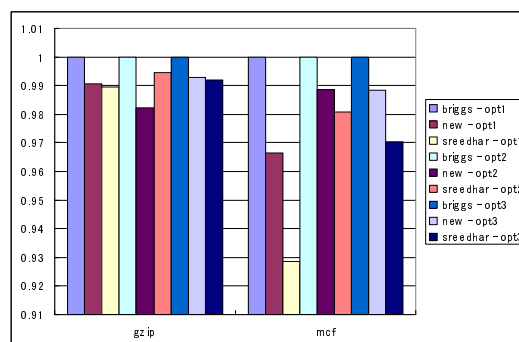


図 12 レジスタ 20 本での実行時間の相対比

Fig. 12 Ratio of execution time in 20 registers

### 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。図から判るように、Sreedhar らの方法は Briggs らの方法と比べて、gzip で 1% 程、mcf では 2 ~ 7% 程速い。

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% 程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判った。また、改良案は若干の効果があることも判った。これらに関する詳しい解析は<sup>13)</sup>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、次の節では 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。

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

表 3 最適化の組み合わせ

Table 3 Combination of optimizations

## 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、前節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。

レジスタの数は前節と同様 8 本と 20 本で行う。最適化の組み合わせは表 3 の通りである。なお、この節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、(レジスタ割り当てにおいて) コアレス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(第 6.1 節)。

コアレス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, レジスタ割り当てでの積極的なコアレスシングと同様の問題点があることも予想される(第 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), レジスタ割り当て時のスピルの数を測定する。これは, コアレスシングできないコピー文が目的コードやレジスタ割り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, コアレスシングできないコピー文の数の差や,  $\phi$  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わ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 本に制限した時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%程だった。これによって, Briggs らの方法ではコアレス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。

最適化の組み合わせによる比較では, 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。

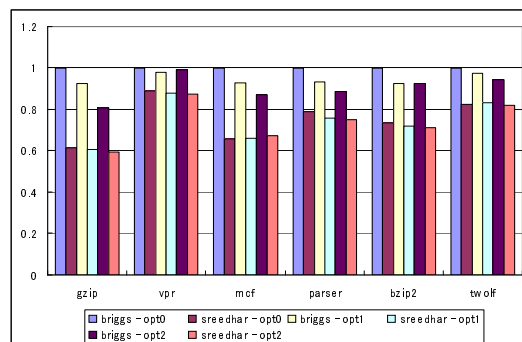


図 13 レジスタ 8 本でのムーブ命令数の相対比

Fig. 13 Ratio of number of moves in 8 registers

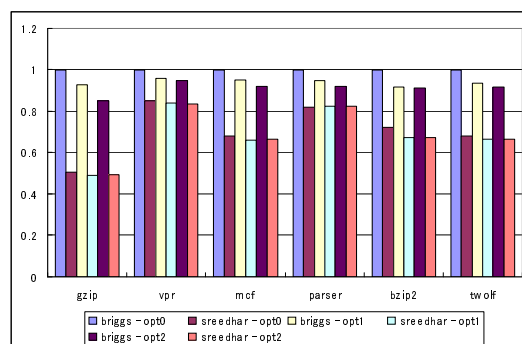


図 14 レジスタ 20 本でのムーブ命令数の相対比

Fig. 14 Ratio of number of moves in 20 registers

図から判るように, 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%程だった。これは, レジスタ数 8 本の時より大きな差となった。

最適化の組み合わせによる比較では, レジスタ数 8 本の時と同様に, 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 本の時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, コアレスシングできないコピー文の数の差や,  $\phi$  関数内の変数を同一化処理することがレジスタ割り当てに与える影響を調べるため, レジスタ割り当て時にスピルされる変数の数を調べる。

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わ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 本に制限した時のレジスタ割り当て時のスピルの数を示す。

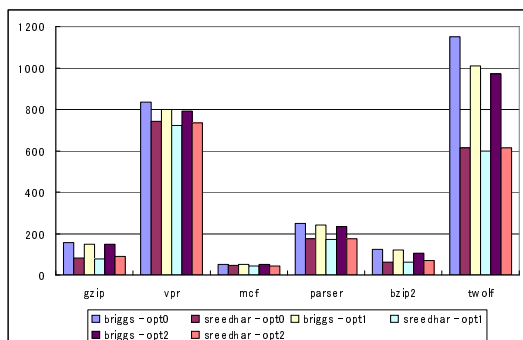


図 15 レジスタ 8 本でのスピル数

Fig. 15 Number of spills in 8 registers

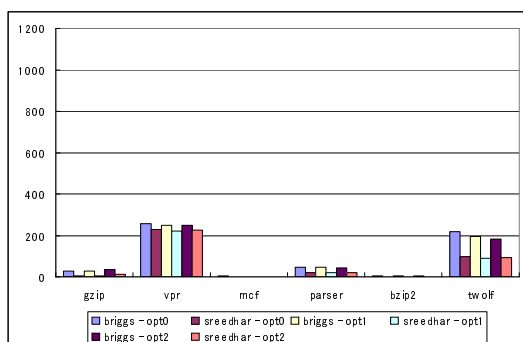


図 16 レジスタ 20 本でのスピル数

Fig. 16 Number of spills in 20 registers

図から判るように、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ス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 本にした時のレジスタ割り当て時のスピルの数を示す。

図から判るように、レジスタ数 8 本のとくと同様に、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 本のとくよりスピルの数は少ない。これによって、レジスタ数 8 本の時よりロード・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、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

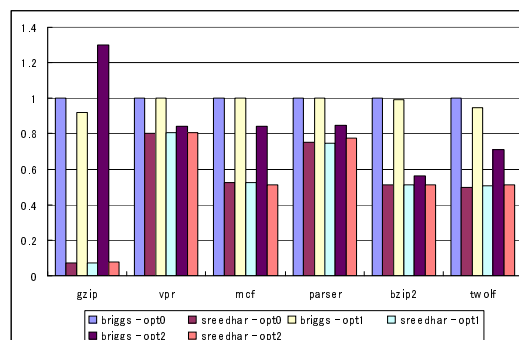


図 17 レジスタ 8 本でのムーブ命令の実行数の相対比

Fig. 17 Ratio of number of execution of moves in 8 registers

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 本に制限した時のムーブ命令の実行数の相対比を示す。

図から判るように、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%程の実行数であった。

最適化の組み合わせによる比較では、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 本にした時のムーブ命令の実行数の相対比を示す。

図から判るように、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%程の実行数であった。これも、静的な測定と同様にレジスタ 8 本の時と比べて差が大きくなっている。

最適化の組み合わせによる比較では、レジスタ数 8 本の時にもまして、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  $\phi$  関数内の変数を同一化しているため、 $\phi$  関数内の変数の生存区間が長くなり、変数一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 本に制限した時のロード・ストア命令の実行数の相対比を示す。

図から判るように、ロード・ストア命令の実行数に

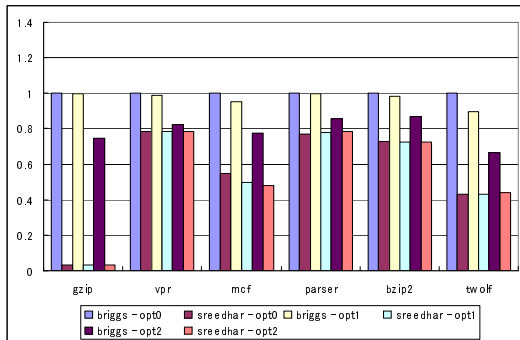


図 18 レジスタ 20 本でのムーブ命令の実行数の相対比  
Fig. 18 Ratio of number of execution of moves in 20 registers

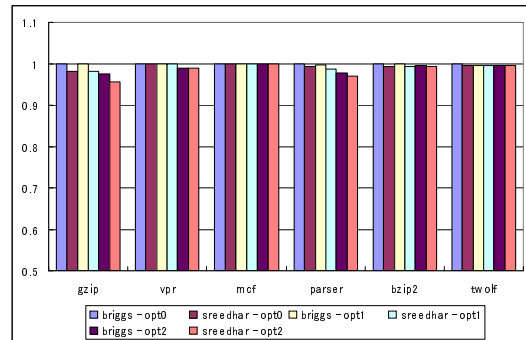


図 20 レジスタ 20 本でのロード・ストアの実行数の相対比  
Fig. 20 Ratio of number of execution of load & store in 20 registers

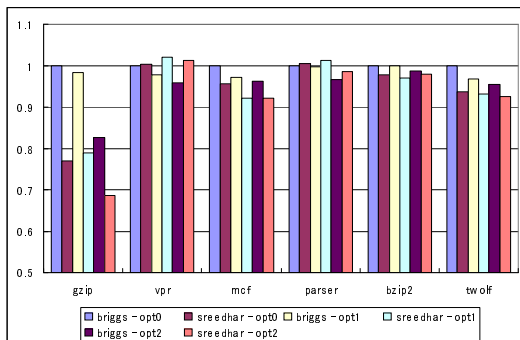


図 19 レジスタ 8 本でのロード・ストアの実行数の相対比  
Fig. 19 Ratio of number of execution of load & store in 8 registers

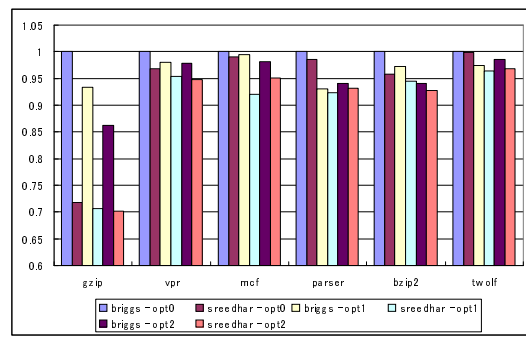


図 21 レジスタ 8 本での実行時間の相対比  
Fig. 21 Ratio of execution time in 8 registers

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、スピルされる変数一つあたりのコストが高いこともわかる。

最適化の組み合わせを変えても、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 本にした時のロード・ストア命令の実行数の相対比を示す。

図から判るように、レジスタ 8 本する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。

### 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。相対比からわかるように、Sreedhar らの方法は Briggs らの方法と比べて、gzip では 28% 程度、その他でも全て若干速い。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 の全ての最適化の組み合わせ、parser の最適化なしでは最大 3% 程遅い。その他では最大 8% 程速い。

実行時間の比較では、概ねここまでの考察のとおり結果であったが、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

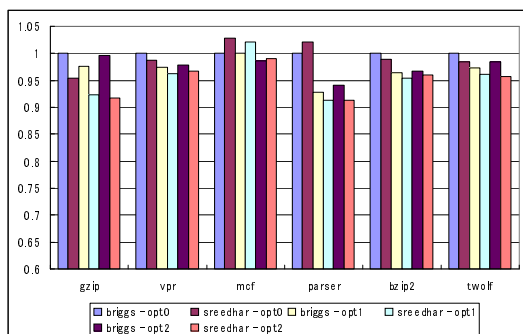


図 22 レジスタ 20 本での実行時間の相対比

Fig. 22 Ratio of execution time in 20 registers

列性の影響によるものであると思われる。原因の一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。例えば、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したが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掛かる。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<sup>13)</sup>。このような遅延は今回実験に用いたようなスーパースカラマシンでは特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、最適化の組み合わせを変えても、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. 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い。そこで、本節ではその他の SSA 逆変換アルゴリズムについて述べる。

### 9.1 Morgan による SSA 逆変換

まず Morgan の逆変換<sup>15)</sup> について簡単に説明する。

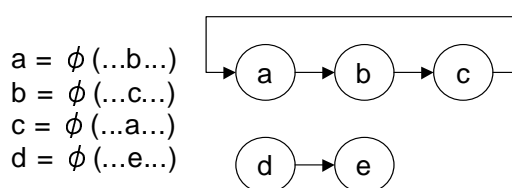


図 23 Morgan の方法

Fig. 23 Method of Morgan

このアルゴリズムは Briggs らと同様に 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、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、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。例えば、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。

### 9.2 Budimlic らによる SSA 逆変換

次に Budimlic らの逆変換<sup>2)</sup> 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。基本的なアイデアは Sreedhar らと同様、 $\phi$  関数を同一化することで逆変換を行なう。このアルゴリズムでは、まず SSA 形式で表わ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  $\phi$  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と思われる。

## 10. 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、検証を行なった。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られる。

得られた知見として、

- Briggs らの方法ではコアレス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、 $\phi$  関数に対してコアレス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%程有利となること。
- 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 らの方法が多くなる場合に実行時間が数%程有利であること。

などが挙げられる。

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, *Proceeding of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp. 25–32 (2002).
- 3) Chaitin, G. J.: Register Allocation & Spilling via Graph Coloring, *Proceedings of the 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 Transactions on Programming Languages and Systems*, 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 Transactions on Programming Languages and System*, Vol.18, No.3, pp. 300–324 (1996).
- 9) Harvard University: MachSUIF Homepage. <http://www.eecs.harvard.edu/machsuirf/>.
- 10) IBM: Jikes Research Virtual Machine. <http://www-124.ibm.com/developerworks/oss/jikesrvm/>.
- 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: Proceedings of the 18th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ACM Press, pp. 187–204 (2003).
- 13) 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).
- 14) 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, 修士論文, 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 (2001).
- 15) Morgan, R.: *Building an Optimizing Compiler*, Digital Press (1998).
- 16) 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).
- 17) Scale Compiler Group: University of Massachusetts. <http://www-ali.cs.umass.edu/Scale/>.
- 18) Sreedhar, V. C., Ju, R. D.-C., Gillies, D. M. and Santhanam, V.: Translating Out of Static Single Assignment Form, *Proceedings of the 6th*

*International Symposium on Static Analysis*,  
Lecture Notes in Computer Science, Vol. 1694,  
Springer-Verlag, pp. 194–210 (1999).

---