SSA形式によるレジスタ割付

1E−4

吉田 順[1]　佐々 政孝[2]

[1] 東京工業大学社会理工学研究科価値システム専攻　[2] 東京工業大学数理計算科学専攻

1 序論

本研究はSSA形式を利用することによって実行効率をあまり落とさずに，レジスタ割付のアルゴリズムを単純化する方法を提示する．また，すべてのアルゴリズムを属性文法で記述することによって，可読性，保守性の高い記述をすることができたので，これもあわせて報告する．

2 SSA形式

2.1 SSA形式について

SSA形式 (Static Single Assignment Form) とはコンパイラ内部の中間形式で，すべての変数への代入を一回に限定するような形式である (図1の (b))．変数への代入を一回に限定することによって，最適化を簡単かつ効率よく行うことができる．代入を一回に限定するために，コントロールフローが合流したときの値をマージする仮想関数，ϕ 関数が存在する．

2.2 ϕ 関数消去の方法

ϕ 関数が存在する状態のままでレジスタ割付を行うことはできない．そこで一旦 ϕ 関数を消去し (**正規化**と呼ばれる)，その後レジスタ割付を行う．ある結合ノードにおける ϕ 関数の消去方法は，その結合ノードの先行者 (predecessor) の一番最後に代入文を挿入することである (図1の (c))．またこの代入文の左辺の変数のことを**正規化された変数**と呼ぶことにする．

3 レジスタ割付のアルゴリズム

レジスタ割付とは，実行速度を上げるためにレジスタをメモリの代わりに使用するということである．これには様々な解析を行わなければならない．しかしSSA形式を利用することで，実装を単純にし，かつ効率の良いレジスタ割付を行うことができる．本研究では言語としてC言語を想定している．また，割付はC言語の関数ごとに行なうものとする．

本割付はレジスタ割付の一般的なやり方を踏襲している。つまり生存区間の解析をし，干渉グラフの作成をし，N-彩色問題を近似的に解き，レジスタを割り当てると言う方法を採っている．
SSA形式を用いる利点は以下のようなものが挙げられる．

- それぞれの変数への代入が一度しかないため，解析がしやすい．
- 制御構造のデータフロー情報が ϕ 関数に集約されているため，ϕ 関数さえ特別扱いすれば，関数ごとのレジスタ割付の実装は単純になる．

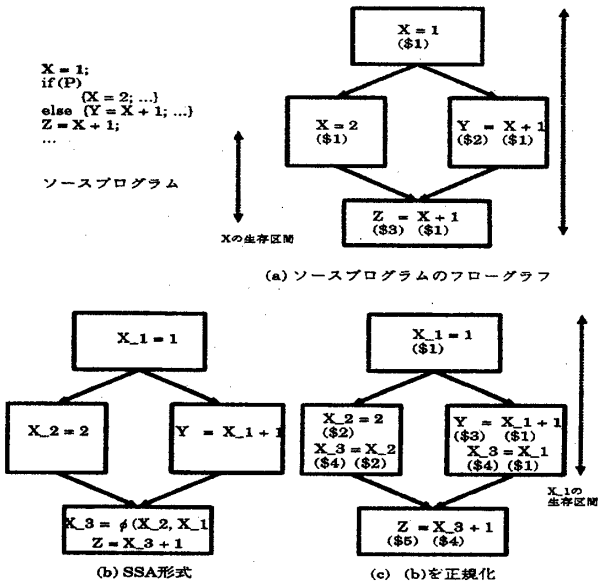

図 1: SSA形式を用いてレジスタに置く例

このため生存区間を求める場合は，

- 関数の下から上に向かってたどり，ある変数が始めて右辺に現れた所から，ただ一つの代入までを求めれば，その変数の生存区間が定まる．

以上により，従来の生存区間を求めるデータフロー解析を行う方法に比べ，レジスタ割付のアルゴリズムは単純化される．例えば先程の図1を見てみよう．図1の (a) では、if文の直前と，then節中に変数 X への代入があったため，それぞれを同じレジスタに置く必要があった．しかしSSA形式を正規化したプログラムとは，「正規化された変数」以外の変数は，今だに単一代入形式を保っているので，同

Register allocation by SSA form
Jun Yoshida[1], Masataka Sassa[2]
[1]Department of Value and Decision Science, Tokyo Institute of Technology,
[2]Department of Mathmatical and Computing Science, Tokyo Institute of Technology

じレジスタに置く必要はない. これにより実装が単純になる. 正規化された変数については同じレジスタに置く必要がある. これは属性文法を利用することにより, 正規化された変数に「正規化された」という情報を属性として残すことによって, 他の変数と区別をした.

またこの後, 干渉グラフを作り彩色をおこなう. 彩色方法についてであるが, Chow のように変数の優先度をつけ register spilling を行なった [Chow]. 変数の優先度の付け方は Chow と異なり, 次式のようにした.

$$\text{優先度} = \text{重み}/\text{生存区間の長さ}$$

ここで重みは次式で表される.

$$\text{重み} = \sum_{I \in expression} 10^{depth(I)}$$

expression は, ソースプログラム中のすべての式の集合を表し, I はその式の中から, その変数を含む一つの式を取ったものである. *depth* はループの入れ子の深さを表す.
生存区間の長さはプログラムの字面のみを見た時の生存区間の長さとした.

4 実装

本研究で行った正規化, レジスタ割付は属性文法による C 言語コンパイラである Cmm のフェーズの一部となっている. コンパイラのフェーズはフロントエンド, SSA 形式変換, 最適化, 正規化, レジスタ割付, コード生成の順となっている. 実装はすべて属性評価器生成系 Jun[Sasaki] を利用して記述した. 属性文法を利用することは, 記述の簡潔さ, 読みやすさなどの長所がある. 割付はソースプログラムからフロントエンドによって変換された抽象構文木上で行っている. 記述の一部を下記に示す.

```
%class
C_STM ::= compound_statement | statement_s
        | statement | if | do | return ...
%semantics
compound_statement => LEXICAL, C_STM;{
compound_statement.live_range_in =
        C_STM.live_range_in;
compound_statement.interfer_graph_in =
        C_STM.interfer_graph_in;
compound_statement.register_allocate_in =
        C_STM.register_allocate_in; ...} ...
```

5 実験比較

簡単なプログラムを用いて gcc の出力と実行効率の比較をしてみた (図 2). この図でいう最適化は定数伝播, ループ不変式の移動, 不要コードの除去である.

6 結論

まとめると本研究の特徴は次のようになる.

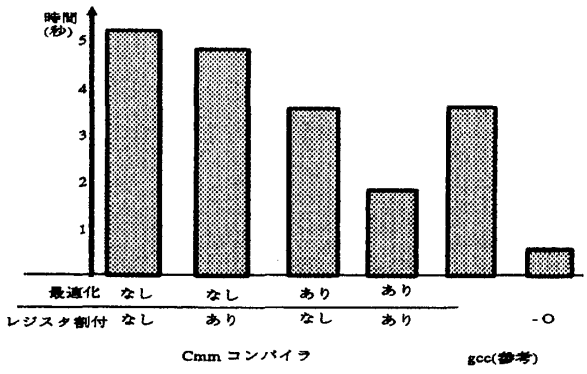

図 2: Multiply の例の実行時間の比較

- SSA 形式により, レジスタ割付のアルゴリズムが単純化される.
- 属性文法により, 記述が簡便になり, かつ可読性が向上した.

つまり単純な実装でも目的コードの効率をかなり高速化できることがわかった. なお本研究の一部は文部省科学研究費の補助を受けた.

7 今後の課題

図 1を見てもわかるとおりソースプログラムでは 3 レジスタを必要としているのに, 本割付では 4 レジスタを必要とする. つまりさらに良い割付があるのは明らかである.

本手法を Chow の割付方法 [Chow], すなわち基本ブロックごとの割付と比べた場合, 基本ブロックのはじめと終わりにできるレジスタからメモリへの移動が, レジスタからレジスタへの移動に移ったものであると考える. つまり実装は単純であるが, Chow の方法より有利であると言える.

ただし今後の課題として, コアレッシング [George] などによりさらに効率の良い割付を進めるつもりである.

参考文献

[Chow] 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, October 1990.

[Sasaki] 佐々木 晃: 循環属性文法に基づく生成系 *Jun,* 日本ソフトウェア科学会, 第 12 回大会, pp. 293-296, 1995.

[George] George,L. and Appel,A.W.: *Iterated Register Coalescing,* ACM POPL. pp.208-218, 1996.