Array SSA とそれを用いた最適化の実装と評価

米倉 翔一　　佐々 政孝

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

静的単一代入 (SSA) 形式とは、変数の定義がプログラムの字面上で唯一となるようにしたプログラムの中間表現の形式である。この SSA 形式の配列への拡張 (Array SSA) は、既存の手法では実用性の低いものであったが、Rus らによって実用的な拡張手法が提案された。この手法は Fortran に対して行われているため、C 言語のプログラムに適用するにはいくつかの変更が必要である。また、Array SSA の効果に対する評価はまだあまり行われていない。そこで本研究では、Rus らの手法を参考に、C 言語に対応した Array SSA と、それを用いた 4 つの最適化を COINS コンパイラ上で実装し、その効果を計った。実験の結果、数%から 10%の実行時間の改善が見られ、その効果を確認することが出来た。

Implementation and Evaluation of Array SSA and its Use in Optimization

Shoichi Yonekura　　Masataka Sassa

Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology

Static single assignment (SSA) form is an intermediate program representation in which each variable has only one definition in the program text. Existing techniques of extentions of SSA form to array variables (Array SSA) were limited in functionality. Rus et al. proposed a technique of practicable Array SSA. Because this technique targets Fortran, some changes are necessary to apply it to programs in C language. In this paper we have implemented the Array SSA applicable to C language using the COINS compiler referring to the techniques of Rus. We also implemented optimizations in Array SSA and their effect. Experimental results show that Array SSA improves the performance of several programs by a few to 10%.

1　はじめに

静的単一代入形式とは、変数の定義がプログラムの字面上で唯一となるようにしたプログラムの中間表現の形式である [2]。SSA 形式では定義と使用の関係が明確になるため、コンパイラにおけるデータフロー解析や最適化を見通しよく行うのに適している。

SSA 形式を配列に適用できるように拡張したものを Array SSA という。配列の定義は、スカラー変数の定義と違い、配列全体を書き換えるとは限らないため、配列に対して SSA を適用することは容易ではない。

Array SSA の過去の手法は Knobe らの手法 [4] などがある。しかし、これらは保守的であったり、実行時にオーバーヘッドがあるなど、実用性の低いものであった。

最近、Rus らによって実用的な Array SSA の手法が提案された [6]。この手法では, 個々の配列要素ではなく、配列要素の領域 (Region) ごとに定義されたときの SSA 名を求める。この手法は Fortran を対象としているため、そのままでは C 言語のプログラムに適用することが出来ない。また、Array SSA の実装はまだあまり行われておらず、そのためその効果もあまり評価されていない。

本研究では、Rus らの手法による Array SSA を C 言語のプログラムに適用できるように拡張し、COINS コンパイラ [1] の高水準中間表現 HIR 上で実装を行った。また、Array SSA を用いた最適化の効果を評価するため、4 つの最適化を実装し、その効果を計った。

2 Array SSA

2.1 Rus らの手法

Rus らの Array SSA[6] は、配列要素を領域で表し、その領域ごとに定義の SSA 名を記憶する。配列要素の領域の表現には **USR** というメモリアクセスの表現形式を利用する。

2.1.1 ϕ 関数の定義

スカラー変数に対する SSA では、制御フローの合流点に ϕ 関数を挿入する。一方 Array SSA では、制御フローの合流点だけでなく、配列の定義の後にも ϕ 関数を挿入する。配列に対する ϕ 関数では、ϕ 関数に配列要素の領域の情報を組み込み、配列の SSA 名とその定義された要素の領域のペアで表現する。

$$[A_n, \mathcal{R}_n] = \phi(A_0, [A_1, \mathcal{R}_1^n], ..., [A_m, \mathcal{R}_m^n]) \quad (1)$$

$$where\ \mathcal{R}_n = \bigcup_{k=1}^{m} \mathcal{R}_k^n\ and\ \mathcal{R}_i^n \cap \mathcal{R}_j^n = \emptyset, \quad (2)$$
$$\forall\ 1 \leq i, j \leq m, i \neq j$$

式 (1) に配列に対する ϕ 関数の一般形を示す。$A_k(k = 1, ..., m)$ は配列 A の SSA 名を表す。$\mathcal{R}_k^n$ は A_k で定義され、この ϕ 関数まで運ばれてきた配列要素の領域、つまり A_k 以後、A_n までの間に上書きされていない要素の領域である。$\mathcal{R}_n$ は A_n で定義されている全配列要素の領域である。A_0 は A_n で未定義な領域を表す。式 (2) により、各 $\mathcal{R}_k^n$ は互いに素になっていることがわかる。この性質により、ある配列参照 $A_n[i]$ で利用される配列要素 i の定義位置が A_0 から A_m のどれかに一意に定まる。

図 1 に配列の定義の後の ϕ 関数の例を示す。(a) のプログラムを Array SSA に変換すると (b) のようになる。ここでは A_1, A_3, A_5 の各定義文の後に、定義の後の ϕ 関数である δ 関数が挿入されている[1]。A_2 では、要素 (配列の添え字のこと){1} が定義された A_1 と未定義を表す A_0 を合成している。A_2 で定義されている要素は A_1 で定義された要素 {1} のみであり、それ以外の要素はすべて未定義となる。A_4 では、直前で定義された A_3 と、それ以前の定義である A_2 を合成している。A_2 で定義されている要素は {1} であり、A_3 では要素 {2} が定義されているので、A_4 で定義されている要素の領域は $[1:2]$ と表せる (A_4 の定義の左辺)。

ループに対する Array SSA の例を図 2 に示す。ループ内の配列の定義では、ループの繰返しごとに定義されている要素の状態が異なる。この繰返しごとの定義を表すために、ループボディの先頭に μ 関数を挿入する (図 2(b) の 3 行目)。μ 関数内の $(i = 1, 10)$ は i がこのループの制御変数 [5] であり、i の値が 1 から 10 まで変化しながらループを反復することを表している。μ 関数の引数内の領域は i の関数となっており、i の値を代入することにより、ループの繰返しごとの定義されている領域が求まる。例えば、$i = 5$ でのループの実行開始時の、配列 A の定義された要素の領域は、$i = 1, ...4$ のときに、A_2 で定義されている $[1:4]$ となる。実際に i に 5 を代入すると A_1 の定義領域は $[1:4]$ となり正しく求まることがわかる。

2.1.2 定義の探索

Array SSA 形式上では、ϕ 関数の引数をたどって行くことにより、使用する配列要素の定義された位置が求まる。例として図 1 の最後の行で使用されている $A_6[2]$ がどこで定義されているのか探索する。A_6 の定義の左辺を見ると、A_6 では領域 $[1:2]$ が定義されているので、$A_6[2]$ は定義されていることがわかる。右辺を見ていくと、要素 {2} を含む領域を定義しているのは A_4 であることがわかる。そこで A_4 の定義文を見てみると、同様に右辺の δ 関数の引数から、要素 {2} は A_3 で定義されていることがわかる。A_3 は ϕ 関数でない実際の定義文なので、$A_6[2]$ の定義位置は A_3 となる。A_3 では 3 が代入されているので定数伝播を行えば、図 1(b) の最後の行は $x_1 = 3$ に変換することが出来る。

2.2 C 言語への拡張

Rus らの手法を C 言語のプログラムに適用するには、次のような問題点がある。

- ポインタ

[1] Rus らの Array SSA では、配列の定義の後の ϕ 関数を δ, ループ出口に挿入される ϕ 関数を η、If ブロックの出口に挿入される ϕ 関数を γ で表す

$A[1] = 1$	$[A_0, \emptyset] = Undefined$
$A[2] = 3$	$A_1[1] = 1$
$A[1] = 2$	$[A_2, \{1\}] = \delta(A_0, [A_1, \{1\}])$
$x = A[2]$	$A_3[2] = 3$
	$[A_4, [1:2]] = \delta(A_0, [A_2, \{1\}], [A_3, \{2\}])$
	$A_5[1] = 2$
	$[A_6, [1:2]] = \delta(A_0, [A_4, \{2\}], [A_5, \{1\}])$
	$x_1 = A_6[2]$

(a) 通常形式　(b) ArraySSA 形式

図 1: Array SSA の例 1

Do $i = 1, 10$	$[A_0, \emptyset] = Undefined$
$A[i] = ...$	**Do** $i = 1, 10$
EndDo	$[A_1, [1:i-1]] = \mu(A_0, (i = 1, 10), [A_3, [1:i-1]])$
	$A_2[i] = ...$
	$[A_3, \{i\}] = \delta(A_1, [A_2, \{i\}])$
	EndDo
	$[A_4, [1:10]] = \eta(A_0, [A_1, [1:10]])$

(a) 通常形式　(b) ArraySSA 形式

図 2: ループに対する Array SSA

- goto 文
- 複雑なループ
- switch 文

本節では、これらの問題に対する本手法での Array SSA の適用方法について述べる。

2.2.1 ポインタ

配列がポインタ参照されている場合、ポインタによって非明示的に配列の要素が書き換えられている可能性がある。本手法では、ポインタ参照されている可能性のある配列に対しては Array SSA を適用しない。

2.2.2 goto 文

プログラム内に goto 文がある場合、goto 文のジャンプ先のブロックでは、制御が goto 文から来たのかどうか分からないため、配列の定義を正確に求めることが出来ない。

本手法では、goto 文のジャンプ先のブロックの先頭に

$$[A_n, All] = Undefined \tag{3}$$

を挿入し、この点で配列の定義が未定義であるということにした。これにより、この文より後ではこのブロック以前の配列の定義を探索することが出来ない (本当は定義されていたとしても未定義と判断される)。しかし、それ以外の部分では、通常通り探索することが出来る。

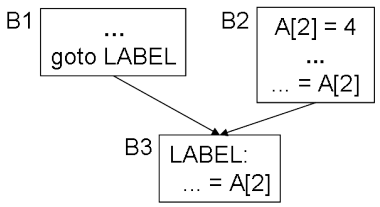

図 3: goto 文を含むプログラムの例

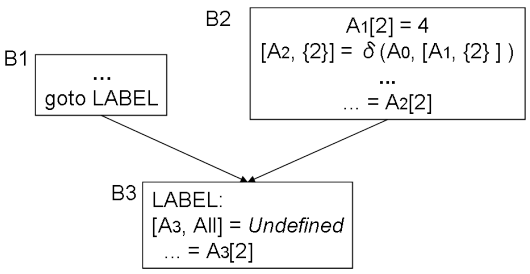

図 4: 図 3 の Array SSA 形式

図 3 に goto 文を含むプログラムの例、図 4 に図 3 のプログラムの Array SSA 形式を示す。このときブロック B3 では、制御が B1 と B2 のどちらから来たのか分からないため、B3 の先頭に式 3 が挿入されている。そのため、B3 で使用している $A_3[2]$ の定義位置は求めることができない。しかし、goto 文と関係がない B2 の最後の行で使用している $A_2[2]$ は、定義を探索していくことにより、$A_1[2]$ で定義されていると求めることがで

きる。

2.2.3 複雑なループ

C 言語には Fortran の Do ループはない。しかし、C 言語の for ループや while ループでも

```
for(i = 0; i < 10; i=i+1){
        A[i] = 1;
}
```

のように、ループの制御変数があり、ループ外からループ内へのジャンプ命令などがない場合は、Do ループと同じように考えることが出来る。

一方で、ループ内に break 文があるとループの出口では、break 文によってループ出口に到達したのか、ループの繰返し条件が成り立たずにループ出口に到達したのかわからない。そのため、ループ出口では配列の定義をうまく求めることが出来ないが、ループ内では Rus らの手法のように解析することが出来る。

また、continue 文がある場合は、continue 文を含むブロックを支配していないブロックの中に配列の定義があると、その定義が実行されるかどうか分からないため、ループの繰返しごとの配列の定義の状態を求めることが出来ない。しかしそのようなものがない場合は、Rus らの手法の通りに Array SSA を適用できる。

ループ外からループ内に進入してくる goto 文などがあると、配列の定義の初期状態が求まらないため、Rus らの手法の Array SSA を適用できない。

以上をまとめて、本手法では以下のようにループを分類して Array SSA を行う。

- 表 1 の 5 条件を満たすループ

  → Rus らの Array SSA をそのまま適用する (μ 関数、η 関数を挿入)

- 表 1 の条件 1〜4 を満たすが、break 文を含むループ

  → ループ内には Rus らの手法を適用するが (μ 関数を利用)、ループ出口には、ループ内で定義されている変数に対して、式 (3) を挿入する

1: for ループまたは while ループ
2: ループの制御変数を持つ
3: ループの繰返し条件がループ制御変数を含む比較式 ($<, >, \leq, \geq$) になっている
4: goto 文などのループ外からループの途中に進入してくるものがない
5: break 文, continue 文を含まない

表 1: ループの条件

- 表 1 の条件 1〜4 を満たすが、continue 文を含むループ

  – continue 文を含むブロックを支配していないブロックに配列の定義は含まれない (continue 文を含むブロックには配列の定義があっても良い)

    → Rus らの Array SSA をそのまま適用する

  – continue 文を含むブロックを支配していないブロックに配列の定義が含まれる

    → ループ内で定義されている配列に対し、ループボディの先頭とループ出口に式 (3) を挿入する

- それ以外

  → ループ内で定義されている配列に対し、ループボディの先頭とループ出口に式 (3) を挿入する

2.2.4 switch 文

switch 文では switch 文の最後か break 文に到達するまでは文を順番に実行していく。そのため、switch 文内の制御フローは複雑になってしまうことがある。本手法では、switch 文内で定義されている配列に対して、switch 文の前後に式 (3) を挿入した。

2.3 適用手順

本手法での変換の手順は以下のようになる。

1. ループの正規化

2. SSA 形式への変換

3. Array SSA 形式への変換

4. 各種最適化

5. SSA, Array SSA から通常形式への逆変換

ループの正規化とは、ループの制御変数が初期値:0、終値: ループの全繰返し数-1、増分値:1 となるようにする変換である。

本手法では、以下の 4 つの最適化を行った。

- 定数伝播
- コピー伝播
- ループ不変式移動
- 配列参照の一時変数化

定数伝播、コピー伝播は、それぞれ定数、スカラー変数が代入されている配列要素の参照を定数、スカラー変数で置き換えるものである。ループ不変式移動は、式の右辺に配列参照が含まれている場合でも不変式移動を可能にするものである。

配列参照の一時変数化は、ループ内で参照されている配列要素が一定のときにループ内での配列参照を一時変数で置き換えるというものである。これにより、ループ内での冗長な配列のアドレス計算を除去することが出来る。例えば図 5 では、(a) の内側のループ内で参照している配列 A の要素は i で一定なので、(b) のようにループの前で一時変数 t に代入し、ループを出た後で $A[i]$ に戻すように変換を行っている。

```
for(i = 0; i < 10; i++){
    for(j = 0; j < 10; j++){
        A[i] = A[i] + ...;
    }
}
```

(a) 最適化前

```
for(i = 0; i < 10; i++){
    t = A[i];
    for(j = 0; j < 10; j++){
        t = t + ...;
    }
    A[i] = t;
}
```

(b) 最適化後

図 5: 配列参照の一時変数化

- **no opt**: 最適化なし
- **scalar**: スカラー変数に対してのみ最適化を適用

  SSA 変換 → 定数伝播 → コピー伝播 → ループ不変式移動 →SSA 逆変換

- **array**: スカラー変数と配列に最適化を適用

  ループの正規化 →SSA 変換 →Array SSA 変換 → 定数伝播 → コピー伝播 → ループ不変式移動 → 配列参照の一時変数化 → SSA, Array SSA 逆変換

実験結果は、図 6 のようになった。これは no opt の目的コードの実行時間を 1 としたときの相対値のグラフである。

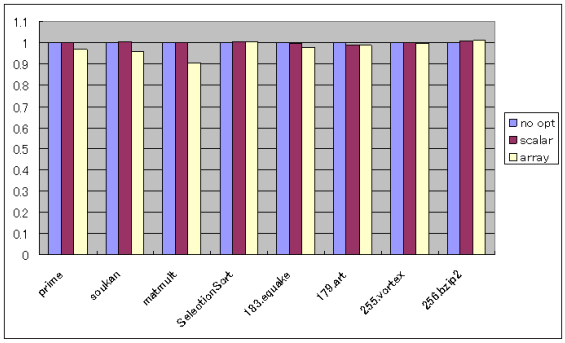

図 6: 実験結果

3 実験

3.1 実行時間

実験には COINS version 1.4.3 を用い、PRIME-POWER250 (プロセッサ SPARC 64 V 1.98GHz ×2、メモリ 10GByte、SunOS 5.10) 上で実行時間を測定した。効果を確認するためのベンチマークプログラムとして、4 個の小さなプログラム (素数算出 (prime)、相関係数の計算 (soukan)、行列の積 (matmult)、選択ソート (Selection Sort)) と、SPEC CINT2000[7] の 4 個のベンチマーク (179.art, 183.equake, 255.vortex, 256.bzip2) を使用して、各最適化を行った。

比較対象は以下の 3 つとした。

図 6 を見ると、小さいプログラムは選択ソートを除いて数%から 10%程度実行時間が改善している。SPEC では 183.equake, 179.art で若干の実行時間の改善が見られた。

そのほかのベンチマークではほとんど効果が出ていない。この原因としては、本手法では最適化の複数回適用や無用命令除去を行っていないことなどが考えられる。

3.2 コンパイル時間

図 7 に SPEC ベンチマークに対する、コンパイル時間を示す。これは no opt のコンパイル時間を 1 としたときの相対値のグラフである。

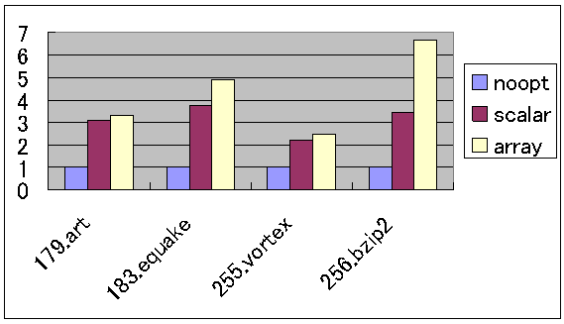

図 7: コンパイル時間

図 7 を見ると、256.bzip2 以外は array のコンパイル時間は no opt の 3〜5 倍に収まっている。256.bzip2 では、no opt の 7 倍近くなっているが、scalar の 2 倍以内に収まっている。以上より、Array SSA は実用的な時間で解析できるといえる。

4 今後の課題

4.1 最適化の繰返し

本手法で行った最適化の複数回適用や、無用命令除去を行うことによって、さらに実行時間が改善されると思われる。今後これらを行って、その効果を確かめたい。

4.2 ループ構造の変換

ループ内での配列参照に対する定数伝播では、参照される配列要素が、ループのすべての繰り返しで、同じ定数で定義されていなければならない。しかし、配列要素の一部分のみが定数で定義されている場合は多い。この場合、ループ分割やループ展開を行うことにより最適化を適用することが出来る。今後これらのループ変換を伴う最適化を行いたい。

5 まとめ

本研究では、Rus らの Array SSA を C 言語のプログラムに適用できるように拡張し、COINS 上で実装を行った。また、Array SSA 上でいくつかの最適化を実装し、その効果を計った。その結果、いくつかのプログラムで数%から 10%の実行時間の改善が見られ、Array SSA が効果的な手法であることを確認できた。

参考文献

[1] COINS Project. COINS home page. http://www.coins-project.org/ .

[2] R. Cytron, et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, Vol. 13, No. 4, pp. 41-486, October 1991.

[3] S.Fink, K. Knobe, and V. Sarkar. Unified Analysis of Array and Object References in Strongly Typed Languages. In *Proc. Static Analysis Symp.*, LNCS 1824, pp.155-174. London, UK, 2000.

[4] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *ACM POPL*, pp. 107-120, San Diego, CA, Jan. 1998.

[5] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.

[6] S. Rus, G. He, C. Alias, and L. Rauchwerger. Region Array SSA. In *ACM PACT'06*, pp. 43-52, Seattle, Washington, USA, Sep. 2006.

[7] SPEC. Standard performance evaluation corporation home page. http://www.spec.org/ .