Array subscript bit vector 表示による依存解析手法*

1J－2

山下 浩一郎† 安田 泰勲† 宮沢 稔‡ 笠原 博徳†

早稲田大学 理工学部† 三菱電機株式会社‡

1 はじめに

自動並列化コンパイラにおいて、ソースコード中の並列性を最大限に引き出すためには強力なデータ依存解析が必要である。特にループにおいてループキャリィドディペンデンス解析 [1][2] を行う場合には、制御変数の値域を考慮しつつ配列変数の添字式を解析しなければならない。データ依存解析の従来手法としては GCD test が最も簡単な手法の一つとして知られているが、ネストしたループにおいて解析対象となる配列変数の添字式 Sub が式 (1) のような複数の制御変数による線形一次式で表現される場合に適用できない。このような場合には Omega test[4] などが使用されるが、最悪の条件下では解析時間のオーダー関数が指数関数となることが知られている。[5]

$$Sub = a_0 + a_1 \cdot I_1 + a_2 \cdot I_2 + \cdots\cdots a_k \cdot I_k + \cdots\cdots + a_n \cdot I_n \quad (1)$$

（ただし I_k は制御変数、a_k は定数係数）

本稿ではこのようなネストループの問題に対して、各々の配列要素にビットを対応させたビットベクトル [6] である Array Subscript Bit vector を用いたデータ依存解析手法を提案する。本手法は単純な論理演算によるネストループにおける複数制御変数線形一次式で表現される依存解析を短い時間で行なうことを可能とする。

2 解析手法

一般にループ内でのデータ依存解析はステートメントの定義／参照変数集合を用いて解析を行なうほかに、配列変数の添字式を考慮したループイタレーション間にわたった解析を行なうことが必要になる。

本節では、提案する Array Subscript Bit vector（ASB ベクトル）の定義、演算法、ならびに実コンパイラへのインプリメンテーションのためのデータ圧縮法を、簡単なネストループ中の配列変数の出力依存解析を例にとりながら説明する。

2.1 ASB ベクトル

図 1の Do ループにおいて配列変数 A に関する ASB ベクトル $V(A)$ について説明する。配列変数 A の定義要素に対応するビットを 1、非定義要素に対応するビットを 0 としたものでベクトルのビット長は配列変数 A の大きさとする（この場合 30）。

```
      REAL A(30)
       ......
      DO 10 I = IL, IU, IS
        A( I + 5 ) = ......
   10 CONTINUE
```

図 1: ASB ベクトル説明のためのサンプルプログラム

ループの初期値、終値、ステップがコンパイル時に分かっている場合、例えば、初期値 IL=1, 終値 IU=19, ステップ IS=3 の場合、式 (2) のように定義する。

$$V(A) = (0000010010010010010010010000000) \quad (2)$$

終値 IU がコンパイル時に値が不明である場合、定義の可能性のある要素のビットを 1 として式 (3) のように定義する。

$$V_{unknownIU}(A) = (0000010010010010010010010001001) \quad (3)$$

また、ステップ IS が不明である場合はこれらの値が負になる可能性を考慮して全てのビットを 1 とし式 (4) のように定義する。

$$V_{allpossible}(A) = (1111111111111111111111111111111) \quad (4)$$

2.2 ビットベクトル演算の表記

次に本 ASB ベクトルを用いたデータ依存解析で使用するビットベクトルに関する演算記号を表 1 に示す。

ここで用いられる演算子でビットベクトルの長さおよび巡回シフトする量は非負整数をとるものとする。また、M, N を自然数、V を ASB ベクトルしたとき、ビットの反復回数は M が $len(V)$ の約数である場合に式 (5) に示す有理数 N/M をとることができる。

$$V \circledast \frac{N}{M} = \left(V \circledast \left\lfloor \frac{N}{M} \right\rfloor\right) \bowtie (V \oslash (N \bmod M)) \quad (5)$$

表 1: 本稿で定義したビットベクトル演算子

記号	意味	表現の例
$\bowtie$	ビットの連結	$(100) \bowtie (101) = (100101)$
$\circledast$	ビットの反復	$(001) \circledast 7/3 = (0010010)$
$\oslash$	左からのビット切り出し	$(100101) \oslash 4 = (1001)$
$\lhd$, $\rhd$	ビット（左, 右）シフト	$(100101) \lhd 2 = (010100)$
$\circledcirc$, $\circledcirc$	ビット（左, 右）巡回シフト	$(100101) \circledcirc 2 = (010110)$
len	ビットベクトル長	$len(100101) = 6$

2.3 ASB ベクトル の圧縮法

Do ループにおいて配列変数の添字式が制御変数の線形一次式である場合には、ASB ベクトルはその内部で同じビットパターンを繰り返しもつ。本手法ではこの性質を利用することにより ASB ベクトルの圧縮を行なう。

具体的にはビットベクトル全体を以下の 3 つの部分に分割する。

- Bit vector の先頭部で繰り返し部分に含まれない部分。($Header$)
- 中間部の同一パターンの繰り返し部分。($Body$)
- Bit vector の後尾部で繰り返し部分に含まれない部分。($Footer$)

このようにして得られた $Header, Body, Footer$ の各部分を用いてビットベクトル V は式 (6) のように表現される。

$$V = Header \bowtie (Body \circledast (\text{繰り返し回数})) \bowtie Footer \quad (6)$$

従ってコンパイラは $Header, Body$ とその繰り返し回数, $Footer$ の各値を確保していればよい。以上のようにしてコンパイラは実際よりも少ないビット数で配列変数の情報を保持することができる。しかし実際にデータ依存解析を行なう異なる 2 つのビットベクトル同士が圧縮された状態であると、各部分のビットベクトル長が等しくない場合がある。このような場合には各部分同士の論理演算が不可能であるため、圧縮された各部分同士を以下に述べる変換方法で長さを揃える必要がある。任意の 2 つの圧縮されたビットベクトルをそれぞれ v_a, v_b とするとき、各々の変換前の $Header$ を H_a, H_b、$Body$ を B_a, B_b、変換後をそれぞれ H'_a, H'_b, B'_a, B'_b のように表す。ここで $len(H_a) \geq len(H_b)$ とすると、比較のために行なう変換は以下のようになる。なお $Footer$ は $Header, Body$ によって表現される配列要素の残りの部分になる。（ここで LCM は最小公倍数を表す）

$$\begin{aligned} H'_a &= H_a \\ H'_b &= H_b \bowtie \left(B_b \circledast \frac{len(H_a)-len(H_b)}{len(B_b)}\right) \\ B'_a &= B_a \circledast \frac{LCM(len(B_a),len(B_b))}{len(B_a)} \\ B'_b &= (B_b \circledcirc (len(H_a) - len(H_b))) \\ &\quad \circledast \frac{LCM(len(B_a),len(B_b))}{len(B_b)} \end{aligned} \quad (7)$$

例えば式 (8) にあげるようなビットベクトル同士の演算を考える。このとき修正前のビットベクトルは式 (9) のように圧縮されている。

$$\begin{aligned} V_a &= (000001001001001001000000) \\ V_b &= (000101010101010101010000) \end{aligned} \quad (8)$$

$$\begin{aligned} H_a &= (00000) \quad B_a = (100) \\ H_b &= (000) \quad B_b = (10) \end{aligned} \quad (9)$$

---

*A data dependence analysis scheme using array subscript bit vector.

†Koichiro YAMASHITA, Yasunori YASUDA, Hironori KASAHARA

†School of science and engineering, Waseda Univ.

‡Minoru MIYAZAWA

‡MITSUBISHI Electric Corporation

$len(H_a) \leq len(H_b)$ より、式 (7) に従って変換を行なうと式 (10) のようになる。

$$\begin{array}{lcll} H'_a & = & H_a & = (00000) \\ H'_b & = & H_b \bowtie ((100)\circledast\frac{5-3}{3}) & = (00010) \\ B'_a & = & B_a\circledast\frac{LCM(3,2)}{3} & = (100100) \\ B'_b & = & (B_b\circledcirc(5-3))\circledast\frac{LCM(3,2)}{2} & = (101010) \end{array} \quad (10)$$

以上のように変換を行なうことによって論理演算を行なうことができる。

2.4 複数制御変数線形一次添字式の解析

本節ではASBベクトルを用いた複数制御変数線形一次添字式の解析についての具体的な処理法について述べる

```
      DO 10 I = 1, IU, IS
        DO 10 J = 1, JU, JS
          A(I+J) = ·····
10    CONTINUE
```

図 2: 出力依存を持つ可能性のある多重ループの例

図2のような多重ループでは配列変数 A の添字式 $I+J$ によって決定される要素の定義順序は IU=7,IS=2,JU=13,JS=3 の場合に図3のようになる。

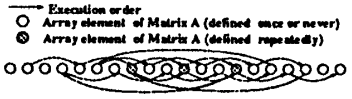

図 3: 実行順序

多重ループにおいて添字式が複数の制御変数によって構成される場合図3に示すように外側の制御変数がインクリメントされるたびに内側のループの制御変数がリセットされるため、各配列要素へのアクセス順序が複雑になる。また、この例では出力依存が生じる要素が存在するためにこのループは Doall 処理が不可能となるが、IS ≤ 5 となるような条件では重複定義要素が存在しないために Doall 可能となる。

このようなループのデータ依存解析は次のようにして行なう。まず外側ループの第1イタレーションについて内側ループに関するビットベクトルを生成しこれを v_1 とする。

$$v_1 = (0100100100100100100100\cdots\cdots) \quad (11)$$

外側のループに関しては v_1 をステップ量で右シフトすることで得られる。このようにして生成されるビットベクトルを $v_2, v_3, \cdots, v_i, \cdots$ とすると v_i は式 (12) のように表すことができる。

$$\left.\begin{array}{lcl} v_2 & = & (0001001001001001001001001\cdots\cdots) \\ v_3 & = & (0000010010010010010010010\cdots\cdots) \\ & & \cdots\cdots \\ v_i & = & v_{i-1} \rhd IS \end{array}\right\} \quad (12)$$

このときループ全体としての定義要素は、これらのビットベクトルの論理和集合として求められるので、これを V_{or} とすると、これは式 (13) のようにする。

$$V_{or}(n) = \bigvee_{i=1}^{n} v_i \quad (13)$$

ここで n は外側ループのイタレーション回数であるが、n が非常に大きい値をもつ場合、もしくはコンパイル時に回数が不明である場合に全ての和集合を計算するのは困難である。そこで、Doall できるかどうかを判断する場合、判別式 (14) によって求めることができる。

$$D = V_{or}(i) \oplus V_{or}(i-1) \quad \text{ただし} \oplus \text{は排他的論理和} \quad (14)$$

式 (14) において $D=0$ であるということは、$V_{or}(i)$ と $V_{or}(i-1)$ が同じビット構成を持つ、すなわち v_i のビットは $V_{or}(i-1)$ において既に立っており、v_iによるオーバーラップにより Doall ができないことを意味する。

次に図3にあるような出力依存要素を漸化的に求める。すなわち、ある v_i において 1 になっているビットが、$V_{or}(i-1)$ にも存在するかを調べればよい。この v_i によって生じるオーバーラップビットベクトルを $V_{Overlap}(i)$ とすると、式 (15) のように求めることができる。

$$V_{Overlap}(i) = V_{or}(i-1) \wedge v_i \quad (15)$$

全体的すなわち厳密解である出力依存要素は $V_{Overlap}(i)\,|_{i=1,2,\cdots,n}$ の論理和集合として式 (16) のように求めることができる。

$$V_{Overlap}(n) = \bigvee_{i=1}^{n} V_{Overlap}(i) \quad (16)$$

2.5 一般的なデータ依存解析

以上の解法を用いて、ステートメントの定義／参照配列変数のASBベクトルを生成することによりフロー依存 (δ), 逆依存 ($\overline{\delta}$), 出力依存 (δ^o) の各依存を求めることができる。ステートメント S_a, S_b が S_aS_b の順番に実行されるとき、そこで用いられる ASB ベクトルを $V(S_a), V(S_b)$ とするとき以下のように求められる。

$$\begin{array}{lll} V_{def}(S_a) \wedge V_{use}(S_b) > 0 & \text{であれば} & S_a\ \delta\ S_b \\ V_{use}(S_a) \wedge V_{def}(S_b) > 0 & \text{であれば} & S_a\ \overline{\delta}\ S_b \\ V_{def}(S_a) \wedge V_{def}(S_b) > 0 & \text{であれば} & S_a\ \delta^o\ S_b \end{array} \quad (17)$$

3 性能評価

本データ依存解析手法は実自動並列化コンパイラにインプリメントされており、その解析に要する時間を図4に示す。なお解析時間計測には SUN SPARC station IPX を用い、OMEGA test に関するデータは文献 [4] によるライブラリ (Copyright ©1994 by the Omega Project) を使用した。サンプルソースプログラムには Omega test ライブラリのテストプログラムを用いた。

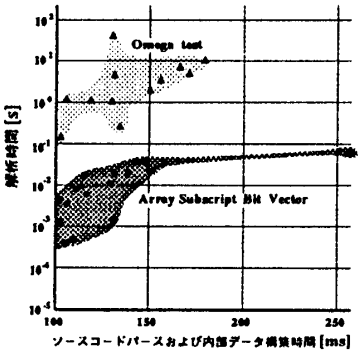

図 4: データ依存解析時間の比較

図の横軸はデータ構築時間を含むプログラムのパースに要する時間、縦軸はデータ依存解析に要した時間を表している。本手法は従来手法である Omega test と比較して解析の対象となる問題の複雑さにかかわらず優れたパフォーマンスを示すことが分かる。

4 まとめ

本稿では ASB ベクトルを使用した配列変数のデータ依存解析を提案し、そのインプリメンテーション手法について述べた。本手法は単純な論理演算を用いて強力なデータ依存解析を高速に行なえることができる。また全ての要素についてビットベクトルを展開せずに、そのパターンが反復される性質を利用して圧縮を行なうことにより実コンパイラへのインプリメントが可能である。今後の課題としては ASB ベクトルの実アプリケーションプログラムへの適用による性能評価があげられる。

本研究の一部は文部省科学研究費 (一般研究 (b)05452354 一般研究 (c)05680284) により行なわれた。

参考文献

[1] Michael J. Wolfe. Optimizing Supercompilers for Supercomputers. MIT press, 1989.

[2] D. J. Kuck etc. Dependence Graphs and Compiler Optimizations.

[3] Utpal Banerjee. Dependence Analysis for Supercomputiong. Kluwer Academic Publishers, 1988.

[4] William Pugh. The Omega Test:a fast and practical integer programming algorithm for dependence analysis. Supercomputing '91

[5] D.Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of presburger arithmetic. Journal of Computer and System Sciences,'78.

[6] Aho, Ullman, etc. Compiler.