软件学报
JOURNAL OF SOFTWARE
1999年 第6期 第10卷 Vol.10 No.6 1999

指针数组的过程内别名分析*

黄 波 臧斌宇 俞一峻 朱传琪

摘要 指针别名分析在C语言的并行优化中占有重要的地位,但已有的指针别名分析只能处理指针标量的情况.文章在介绍已有指针别名信息表示法的不足的基础上,提出了一种能够表示指针数组别名信息的表示方法,它可以更加准确地表示指针别名信息.在此表示法的基础上,提出了指针数组的过程内别名分析算法.此算法完全包含了指针标量的别名分析,对现有的指针别名分析算法所不能解决的一些问题能进行有效地处理.

关键词 指针别名分析,指针数组,过程内分析,并行优化.
中图法分类号 TP314

Intraprocedural Alias Analysis for Pointer Array

HUANG Bo ZANG Bin-yu YU Yi-jun ZHU Chuan-qi

(*Institute of Parallel Processing Fudan University Shanghai* 200433)

Abstract Pointer alias analysis plays an important role in the parallelizing optimization of C program, however, all the previous analyzing algorithm can only be used to analyze the pointer scalars. In this paper, an extended representation for point-to information is presented, which can represent not only the point-to information of pointer scalars, but also the point-to information of pointer arrays. Furthermore, an algorithm of intraprocedural alias analysis for pointer array is presented. This algorithm comprises both the pointer scalar analysis and pointer array analysis. It can be used to efficiently solve some problems which couldn't be solved by the previous algorithm.

Key words Pointer alias analysis, pointer array, intraprocedural analysis, parallelizing optimization.

在过去的10年中,并行优化技术的研究取得了很大进展.国内外的学者先后研制出不少典型的并行优化系统,其中颇具代表性的系统有斯坦福大学的SUIF[1]、伊利诺伊大学的Polaris[2]和复旦大学的AFT[3]等.从各种媒体发表的论文及系统测试数据都表明，对串行FORTRAN程序的并行优化已趋于成熟,目前研究的方向正逐步转向对串行C程序的并行优化.与FORTRAN相比,C语言存在许多FORTRAN所不具有的语法特点,如多级指针的使用、内存的动态分配、递归函数的调用等.这些区别决定了对C程序的并行优化必然与对FORTRAN程序的并行优化有显著的不同,而正是由于这些区别的存在,在一定的程度上无疑加剧了C程序并行优化的难度.

当程序里面存在指针数据类型或者存在通过传送地址方式进行的过程调用时,两个或两个以上的表达式就可能代表同一内存地址,这时，我们称这些代表同一内存地址的表达式互为别名.由于指针数据类型的存在而引起的别名称为指针别名,指针别名的存在是C语言的典型特征之一.当对某一指针别名信息不明确的时侯,所有的数据流分析及相关性分析都必须采用保守的估计,即假设此指针指向所有变量,这种假设必然给分析的准确性带来很大的影响并最终使并行优化的效果大大降低.为了减小指针别名的存在对并行分析与并行变换的影响,加强指针别名的分析是获取更加准确的指针别名信息、提高数据流分析与并行优化效果的有效措施.国内外并行优化的研究者曾提出过多种关于指针别名分析的方法[4~11].但是,迄今为止，所有的指针别名分析方法都只能对指针标量进行别名分析,当碰到指针数组时,已有的方法就无能为力了,如对于下面这个程序段:

```
int (*FunctionPointer[])()={Function_1,Function_2,...,Function_N};
```

当出现诸如*FunctionPointer*［*i*］()的调用,并且i的值可以确定时,由于已有的指针别名分析方法不能取得*FunctionPointer*的指向信息,在分析时就不能确定具体调用的是哪个函数,从而在其他数据流分析(特别是过程间分析)时必须采用保守的估计方法,这必然影响分析的准确性.为此,本文提出了一种新的指针别名信息的表示法.它把指针的范围与所指目标的范围融合于指针别名信息内,能够表示指针数组的别名信息.在这种表示法的基础上,本文提出了指针数组的过程内别名分析算法,进而把指针别名分析从指针标量推广到指针数组,能为C程序的并行分析与并行变换提供更为准确的指针别名信息.

本文第1节在介绍相关的指针别名信息表示法的基础上提出了一种指针别名信息的扩展指向表示法.第2节介绍过程内指针数组别名分析的基本框架.第3节介绍了如何对C程序中的各种语句进行指针数组别名分析的方法.第4节用3个例子阐述了本文所提算法的应用.第5节是总结.

1 指针别名信息的扩展指向表示

1.1 相关的工作

常见的指针别名信息表示法有两种:别名对(alias pair)表示法［4,5］和指向(point-to)表示法［6～10］.别名对表示法是把互为别名的表达式用一个二元组来表示.由于这种表示法将产生大量的信息冗余且信息量太少,后来又采用了指向表示法,即用由一个指针标量与这个指针标量所指向的目标变量构成的二元组表示别名信息,Maryam Emami等人扩展了指向表示法,在二元组的基础上增加了一个确定指向或可能指向标志(*D*或*P*),从而使得别名信息能被更准确地表示与使用［10］.但当某一指针标量指向的目标是数组时,原来的指向表示法要么把整个数组当作一个整体来看待,要么只区分数组的首元素,并把除首元素外的其他数组元素当成一个整体.后来,斯坦福大学的Robert P. Wilson等人用一个位置集(location set)来表示指针所指向的目标,更准确地表示了指针指向的范围(位置集是一个三元组(*b*,*f*,*s*),*b*是内存块的名字,*f*是在*b*内的偏移,*s*是步长,(*b*,*f*,*s*)等价于位置集合{*f*+*i*×*s*|*i*∈*Z*})［9］.图1用一个简单的例子来说明这些指针别名信息表示法的区别(在下例中,假设S_1前的别名信息为空,整数占2字节). **

```
int *q, **p, a[10], i;
...
if (i<0 || i>=10)
    s1: q=&(a[0]);
else
    s2: q=&(a[i]);
    s3: p=&q;
    ...
```

指针别名的表示方法	S_3后的别名信息
别名对表示法	(*q,a),(*p,q),(**p,*q)
一般的指向表示法	(q,a),(p,q)
Maryam Emami提出的指向表示法	(q,a_head,P),(q,a_tail,P),(p,q,D)
Robert P. Wilson提出的指向表示法	(q,(a,0,0),P),(q,(a,0,2),P),(p,(q,0,0),D)

图1　几种指针别名信息表示法的比较

但是,上述这些指针别名信息的表示方法都不能表示指针数组的别名信息,因此在对C程序并行优化的过程中遇到指针数组时必须采取保守的估计,这必然会影响并行优化的效果.为了表示指针数组的别名信息,必须采用一种新的指针别名信息的表示法.

1.2 扩展指向表示法

本文采用一种扩展的指向表示法来表示指针别名信息,这种表示法具有如下的形式:(*p*,*t*,*c*,*PR*,TR).其中*p*表示指针变量名或者指针数组名;*t*表示*p*所指的目标变量名或者目标数组名;*c*表示指针的指向关系是确定指向还是可能指向,取值为*D*或*P*,*D*表示确定指向,P表示可能指向,同时定义*P*与*D*之间的汇合算子∧,∧具有如下性质:*D*∧*D*=*D*,*D*∧*P*=*P*,*P*∧*D*=*P*,*P*∧*P*=*P*.*PR*与*TR*的含义如下:当*p*是数组名时,PR表示*p*所包含的数组元素的范围,其他情况下,*PR*=∅;当t是数组名或动态申请的一大块连续内存区域名(动态申请的内存区域的命名规则见下文)时,TR表示t所包含的数组元素或动态申请的内存区域内元素的范围;其他情况下,*TR*=∅.

*PR*与*TR*可以表示为{(S_i,I_i,E_i)|1≤*i*≤*n*}.在三元组(S_i,I_i,E_i)中,S_i表示区域的起始位置,E_i表示区域的终止位置,I_i表示区域递增的步长,它们都是整数,且以字节为计算单位.也就是说，集合*PR*与*TR*的元素本身也是集合,而且各元素集合之间不相交.因此，*PR*与*PR*之间或*TR*与*TR*之间的运算必须包括它们各自的元素集合之间的运算.

设某一数组元素所占的字节数为*ES*,三元组(*S*,*I*,*E*)所表示的区域实际上相当于如下的字节集(其中*B*表示数组首地址):

{*B*+*S*,*B*+*S*+1,...,*B*+*S*+*ES*-1;*B*+*S*+I,*B*+S+*I*+1,...,
B+*S*+*I*+*ES*-1;...,*B*+*E*-*ES*+1,...,*B*+*E*-1,*B*+*E*}

定义**1**. 设*R*为*PR*或*TR*,则数组的第1个元素R_0可表示为{(0,*ES*,*ES*-1)},第*i*+1(*i*>0)个元素R_i可表示为{(*i***ES*,*ES*,(*i*+1)**ES*-1)}.

定理**1**. 用定义1中的表示法表示R_i与R_j,*j*>*i*,则$R_i \cup R_j$={(*i***ES*,(*j*-*i*)**ES*,(*j*+1)**ES*-1)}（证明从略）.

推论**1**. 若i_n-i_{n-1}=i_{n-1}-i_{n-2}=...=i_1-i_0,则$R_{i0} \cup R_{i1} \cup ... \cup R_{in}$={($i_0$**ES*,($i_1$-$i_0$)**ES*,($i_n$+1)**ES*-1)}.

推论**2**. 对于数组A［*n*］,整个数组区域$R_0 \cup R_1 \cup ... \cup R_{n-1}$可表示为{(0,ES,*n***ES*-1)}.

上述的定理和推论在进行数组范围区域的合并操作时,在很大程度上将会简化结果范围区域的表示.

定义**2**. 设*R*=(S,*I*,E),R'=(S′,*I*′,E′),$R \cap R'$= 当且仅当对属于*R*的任意子区域 $\bigcup_{k=0}^{ES-1}\{B+S+i*I+k\}$ 及属于R'的任意子区域 $\bigcup_{k'=0}^{ES'-1}\{B'+S'+j*I'+k'\}$,恒有

$$\left(\bigcup_{k=0}^{ES-1}\{B+S+i*I+k\}\right)\cap\left(\bigcup_{k'=0}^{ES'-1}\{B'+S'+j*I'+k'\}\right)=\varnothing.$$

表1是对一些指针赋值语句(指针赋值语句的概念将在第3.1节定义)的指针别名信息的表示(为简化描述,数组首地址假设为0).

表**1** 一些指针赋值语句的指针别名信息的表示

语句	指针别名信息
*int a,*p;p*=&*a*	{(*p*,*a*,*D*,∅,∅)}
void *q*();void(*f)();f=*q*;	{(*f*,*q*,*D*,∅,∅)}
*int*p* ［10］,*a*;*p* ［0］=&*a*;	{(*p*,*a*,D,PR_0,∅)},其中PR_0={(0,4,3)}
int a ［100］,*b*,**q*; if (*a* ［0］＞*b*) *q*=&*b*; else *q*=*a*;	{(*q*,*b*,*P*,∅,∅), (*q*,*a*,*P*,∅,TR_0)} 其中TR_0={(0,2,1)}
struct STTYPE{*int a*,*b*;float *f* ［N］;}*S*; *float*p*;*p*=&s.f ［10］;	{(*p*,*s.f*,*D*,∅,TR_{f10})} 其中TR_{f10}={(40,4,43)}
*int*p* ［10］,*q* ［N］;*P* ［0］=*q*+i;	{(*p*,*q*,*D*,PR_0,TR_i)} 其中PR_0={(0,4,3)},TR_i={(2i,2,2i+1)}

C程序中不可避免地存在内存的动态申请,必须通过命名的方式对这些动态申请的内存区域进行标识,才能在指针别名信息里表明哪些指针指向它们.在这里采用一种简单的命名方法,即用动态申请内存语句所在的文件名与此语句所在的行号及循环迭代变量(如果此动态申请内存语句在循环体内)来标识.实际上,当申请一块连续的内存区域像数组一样使用时,在进行指针别名的分析时，我们把这种动态申请的内存区域同数组一样看待.

2 指针数组过程内别名分析的框架

过程内的指针别名分析就是通过对过程内的所有语句依次进行别名分析,最后求出在过程出口处的指针别名信息.这里没有讨论过程调用语句对指针别名信息的影响,实际上，跨过程求取指针别名信息是很重要的,文献［5,6,9～11］对此进行了深入的研究.有了过程内指针别名分析的框架,利用各种

形式的过程间分析技术就不难把指针别名分析推广到过程间分析上.由于前文提出的指针别名信息的表示法既能很好地表示指针数组的别名信息,又能把指针标量别名信息的表示囊括其中,因此，本文介绍的指针数组过程内别名分析的框架实际上同时进行了指针数组与指针标量的别名分析.在某种意义上讲，指针标量是指针数组的一个特例,本文所描述的指针数组的别名分析在概念上完全蕴含了指针标量的别名分析.

在介绍指针数组过程内别名分析的基本框架前,先介绍几个基本概念.

对于给定的程序控制流图CFG及程序内的任一语句S,定义*A*-IN(*S*) 为到达语句*S*的入口之前的指针别名信息集(AliasSet);*A*-OUT(*S*)为离开语句S的出口之后的指针别名信息集;*PRED*(*S*)为控制流图中*S*的所有前驱语句所构成的集合.有了上述定义及上一节所介绍的指针别名信息的表示法,指针数组的过程内别名分析可以用如下的框架来表示:

```
for each statement S in the procedure{
   A-IN(S)=∅;
   A-OUT(S)=StmtAnalysis(S,∅,CFG);
}
do{
   Changed=FALSE;
   for each Statement S in depth first order{
      OldA-IN(S)=A-IN(S);
      A_IN(S)=Merge( ⋃ p∈PRED(S) A_OUT(P));
      if (OldA-IN(S)≠A-IN(S)) {
            Changed=TRUE;
            A-OUT(S)=StmtAnalysis(S,A-IN(S),CFG);
      }
   }
}while  Changed
```

在以上的指针数组过程内别名分析的基本框架中,函数StmtAnalysis完成对语句S内指针数组与指针标量别名信息的分析,它的具体实现将在第3节介绍.Merge实现的是对指针别名信息的合并,并以合并后的指针别名信息集作为函数的返回值,Merge所实现的具体操作如下.

算法. 指针别名信息的合并算法.

```
AliasSet  Merge(AliasSet AS)
{
   for each Pointer variable p exists in element of AS {
      if ((p,x,D,PR1,TR1)∈AS and (p,y,D,PR2,TR2)∈AS and
         ((PR1=∅ and PR2=∅) or PR1∩PR2≠∅) and x≠y){
            change (p,x,D,PR1,TR1) to (p,x,P,PR1,TR1);
            change (p,y,D,PR2,TR2) to (p,y,P,PR2,TR2);
         }
      if ((p,x,P,PR1,TR1)∈AS and (p,y,D,PR2,TR2)∈AS and x≠y)
            change (p,y,D,PR2,TR2) to (p,y,P,PR2,TR2);
      if ((p,x,c1,PR1,TR1)∈AS and (p,x,c2,PR2,TR2)∈AS){
            PR=PR1∪PR2;
            TR=TR1∪TR2;
            delete (p,x,c1,PR1,TR1) from AS;
            delete (p,x,c2,PR2,TR2) from AS;
            if (PR==∅ && TR==∅)
                  c=c1∧c2;
            else
                  c=P;
            AS=AS∪{(p,x,c,PR,TR)};
         }
      }
      return AS;
```

}

在指针别名信息的合并算法中,最后一种合并情况是把多个别名信息简化成一个别名信息,从而可以缩减指针别名信息集中的元素个数.但当*p*为指针数组,*p*所指的目标亦为数组,并且*p*中不同的元素指向的数组是同一数组时,这种合并可能会带来一些指针信息的不精确性.由于实际程序中这种情况比较罕见,因而在进行指针数组的别名分析时发生这种合并的机会很少,对整个分析的影响也就很小.

3 的指针数组别名分析

3.1 指针赋值语句

过程内分析所涉及到的对指针别名信息的改变主要体现在指针赋值语句上.指针赋值语句的定义如下.

定义**3.** 当赋值语句左部表达式的类型求值结果为指针类型时,此赋值语句称为指针赋值语句.

指针赋值语句的执行结果是使赋值语句左部代表的指针变量(指针数组或指针标量)所指向的目标发生改变,从而使指针别名信息发生相应的变化.

定义**4.** 由指针赋值语句的左部所代表的指针变量所构成的集合称为左目标集,记作*L-Target*(*S*);指针赋值语句的右部表达式所指向的目标集合称为右目标集,记为*R-Target*(*S*).

左目标集与右目标集中的元素用三元组(*t*,*c*,*R*)表示.在左目标集中,*t*为指针变量名或指针数组名,而在右目标集中,t可以是任意的变量名(包括对动态申请的内存区域的命名).*c*表示*t*是确定目标还是可能目标,取值和运算性质与第1节指针别名信息表示中的*c*相同,*R*的含义与表示和*PR*或*TR*相似,当*t*为数组名或动态申请的内存区域名时,*R*表示目标包含的范围,其他情况下为$\varnothing$.强制类型转换并不改变左目标集和右目标集,常见的指针赋值语句中左目标集与右目标集的计算见表2(第1栏中表达式的求值结果为指针变量或内存区域的地址,*IMPOSSIBLE*表示指针赋值语句不可能以第1栏中的表达式作为左部,*offset*(*f*)表示域*f*在结构内的偏移量,以字节为单位,*ES*表示数组元素所占的字节数),表中未列出的情况可以通过综合此表中的情况而得出相应的左目标集与右目标集.

表**2** 指针赋值语句的左目标集与右目标集

指针赋值语句 *S*的左(右)部	*L_Target* (*S*)	*R_Target*(*S*)
&*a*	*IMPOSSIBLE*	{(*a*,*D*,$\varnothing$)}
&*a.f*	*IMPOSSIBLE*	{(*a.f*,*D*,$\varnothing$)}
&*a*[*i*]	*IMPOSSIBLE*	{(*a*,*D*, R_i)}
a(*a*为指针标量)	{(*a*,*D*,$\varnothing$)}	{(*x*,*c*,*TR*)|(*a*,*x*,*c*,*PR*,*TR*)∈A_IN(*S*)}
a.*f*	{(*a.f*,*D*,$\varnothing$)}	{(*x*,*c*,*TR*)|(*a.f*,*x*,*c*,*PR*,*TR*)∈A_IN(*S*)}
a[*i*] (*a*是指针数组)	{(*a*,*D*,R_i)}	{(*x*,*c*,*TR*)|(*a*,*x*,*c*,*PR*,*TR*)∈A_IN(*S*),R_i⊆*PR*}
**a* (此时*a*是一个 二级指针变量)	{(*x*,*c*,*TR*)|(*a*,*x*,*c*,$\varnothing$,*TR*)∈A_IN(*S*)}	{(*y*,c_1∧c_2,TR_2)|(*a*,*x*,c_1,$\varnothing$,TR_1)∈A_IN(*S*);(*x*,*y*,*c* IN(*S*);TR_1⊆PR_2}

(**a*).*f*
(*a*是一指向结构
的指针,结构中
有一*f*域,域*f*
是一指针)
	{(*y*,*c*,TR')|(*a*,*x*,*c*,$\varnothing$,*TR*)∈A_IN(*S*);
	if ($TR\neq\varnothing$)/*此时*x*为一结构数组*/
	y=*x*;TR'={(S',*I*,E')|S'=*S*+*Offset*(*f*),
	E'=*E*+*Offset*(*f*),(*S*,*I*,*E*)∈*TR*,}
	else
	y=*x.f*;TR'=$\varnothing$;
	}
		{(z,c_1∧c_2,TR_2)|(*a*,*x*,*c*,$\varnothing$,TR_1)∈A_IN(*S*);
		if ($TR_1\neq\varnothing$) /*此时*x*为一结构数组*/
		y=*x*;TR'_1={(S',*I*,E')|S'=*S*+*Offset*(*f*),
		E'=*E*+*Offset*(f),(*S*,*I*,*E*)∈TR_1}
		else *y*=*x.f*;TR'_1=$\varnothing$;
		(*y*,*z*,c_2,PR_2,TR_2)∈A_IN(*S*);TR'_1⊆PR_2}

(**a*)[*i*] (*a*是一指向指针 数组的指针)	{(*x*,*c*,TR')|(*a*,*x*,*c*,$\varnothing$,*TR*)∈A_IN(*S*); TR'={(S',*I*,E')|S'=*S*+*i***ES*, E'=*E*+*i**ES,(*S*,*I*,*E*)∈*TR*}}	{(*y*,c_1∧c_2,TR_2)|(*a*,*x*,*c*,$\varnothing$,TR_1)∈A_IN(*S*), TR'_1={(S',*I*,E')|S'=*S*+*i***ES*,E'=*E*+*i**ES,((*x*,*y*,c_2,PR_2,TR_2)∈A_IN(*S*),TR'_1⊆PR_2}

p+*N*	*IMPOSSIBLE*	$\{(x,c,TR')\mid(p,x,c,PR,TR)\in A\_IN(S),$ $TR'=\{(S',I,E')\mid S'=S+N*sizeof(*p),$ $E'=E+N*sizeof(*p),(S,I,E)\in TR\}\}$
NULL	*IMPOSSIBLE*	$\{(NULL,D,\varnothing)\}$
malloc(*Type*)	*IMPOSSIBLE*	$\{(MEM\text{-}ALLOC\text{-}NAME,D,\varnothing)\}$
malloc(*Type**N)	*IMPOSSIBLE*	$\{(MEM\text{-}ALLOC\text{-}NAME,D,R_0)\}$

在上表中,*MEM-ALLOC-NAME*为一个宏,它根据动态申请内存语句所在的文件名、行号及循环迭代变量(如果此动态申请内存语句在循环体内)等信息返回一个标识名.

3.2 指针赋值语句的指针数组别名分析

给定程序控制流图*CFG*、指针赋值语句*S*及*S*入口前的指针别名信息集A_IN,指针赋值语句的指针别名分析算法可描述如下:

AliasSet PointerAssignmentAnalysis(*AssignStmt S,AliasSet A_IN,Graph CFG*)

{ $A\_KILL=\{(p,t,c,PR,TR)\mid(p,D,PR')\in L\_Target(S),(p,t,c,PR,TR)\in A\_IN\}$;

$A\_CHANGE=\{(p,t,D,PR,TR)\mid(p,P,PR')\in L\_Target(S),(p,t,D,PR,TR)\in A\_IN\}$;

$A\_GEN=\{(p,t,c_1\wedge c_2,PR,TR)\mid(p,c_1,PR)\in L\_Target(S),(t,c_2,TR)\in R\_Target(S)\}$;

$CHANGED\_SET=(A\_IN-A\_CHANGE)\cup\{(p,t,P,PR,TR)\mid(p,t,D,PR,TR)\in A\_CHANGE\}$;

return Merge($(CHANGED\_SET-A\_KILL)\cup A\_GEN$)

}

3.3 其他语句的指针数组别名分析

虽然C语言包含多种语句结构,但经过程序结构化及规范化后,所有的控制结构都可以归结为顺序、条件、循环3种.在对任一语句进行指针别名的分析时,根据此语句所属的类别(属于简单语句、顺序语句、条件语句、循环语句中的哪一种),分别调用不同的指针数组别名分析程序,对语句进行指针别名分析(由于所有具有副作用的表达式都在程序规范化过程中被等价的表达式(或语句)所替代,指针别名信息不会被规范化后的任何表达式所改变).在过程内分析中,只有指针赋值语句才能对指针别名信息进行修改,因而在对简单语句(如赋值语句、break语句、continue语句等)进行指针别名分析时,只要判断此语句是不是指针赋值语句,如果不是指针赋值语句,则语句出口处的指针别名信息将与语句入口处的指针别名信息一致,否则调用指针赋值语句的指针数组别名分析算法计算指针别名信息.而对于顺序、条件、循环3种控制结构,它们的指针数组过程内别名分析的框架与参考文献［10］中的分析框架相似.

4 应用举例

先看下面这个程序段:

```
    int a［M］,*P,*q,i;
    ...
    p=a;
    q=a+1;
    i=0;
    while (i＜N){
s1: *p=...;
s2: ...=*q;
    ...
s3: p=p+2;
s4: q=q+2;
    i=i+1;
    }
    ...
```

经过本文所描述算法的指针别名分析,在循环体的每个迭代中,指针别名信息为{($p,a,D,\varnothing,TR$), ($q,a,D,\varnothing,TR'$)},其中TR={($4i$,2,$4i$+1)},TR' ={($4i$+2,2,$4i$+3)},0≤i≤N-1.由于$TR \cap TR'=\varnothing$,因此在进行相关性分析时,可以通过对指针别名信息的利用判断出s_1和s_2不存在任何相关性,从而可以通过消除s_3与s_4引起的规约相关把这个循环变换成并行循环.

回到本文前言中所举的那个例子,在语句:int (*FunctionPointer［］)()={Function_1,Function_2,...,Function_N}; 后,函数指针数组的别名信息如下:{(FunctionPointer,Function_1,D,PR_0,$\varnothing$), (FunctionPointer,Function_2,D,PR_1,$\varnothing$),..., (FunctionPointer,Function_N,D,PR_{N-1},$\varnothing$)}, 当碰到FunctionPointer［i］()的调用,并且i的值可以确定时,如i=1,通过FunctionPointer与PR_1查指针别名信息表便可以确定这是对 Function_2的调用,于是可以准确扩展原来保守估计的调用图,从而使得分析程序可以进入Function_2跨过程获取数据流信息.

最后再分析下列程序实例.

```
typed struct DataStruct{
            KEYWORD key;
            CONTENT content;
            struct DataStruct* next;
        }DS;
DS*DataBase［N］,q;     int i;
...
/*Construct the data table*/
i=0;
while (i<N){
s1:   DataBase［i］=q=NULL;
      ...
      while (...){
        ...
        if (q==NULL)
s2:       DataBase［i］=q=(DS*)malloc(sizeof(DS));
        else
s3:       q->next=(DS*)malloc(sizeof(DS));
        ...
        if (q->next!=NULL)
s4:       q=q->next;
        i++
      }
}
...
```

通过前文所述算法的分析可得如下的指针别名信息:{(DataBase,NULL,P,PR_i,$\varnothing$), (DataBase,MEM_ALLOC_s_2_i,P,PR_i,$\varnothing$), (q,NULL,P,$\varnothing$,$\varnothing$), (q,MEM_ALLOC_s_2_i,P,$\varnothing$,$\varnothing$), (q,MEM_ALLOC_s_3_i,P,$\varnothing$,$\varnothing$), (MEM_ALLOC_s_2_i.next,MEM_ALLOC_s_3_i,P,$\varnothing$,$\varnothing$), (MEM_ALLOC_s_3_i.next,MEM_ALLOC_s_3_i.next,P,$\varnothing$,$\varnothing$)}(0≤i≤N-1).由此可知,对于外重循环的不同循环迭代,DataBase与q所指的目标集不相交.通过数组私有化技术可以判断出:指针标量q在循环内是私有变量,DataBase在循环内是私有数组,因此,外重循环可以变换成并行循环.

从上面3个例子可以看出:本文所提出指针数组的过程内别名分析算法能对C程序中指针标量与指针数组的别名信息有效地进行分析,从而在很大程度上提高了并行优化的效率.

5 总 结

本文用五元式扩展了指针别名信息的指向表示法,把指针标量与指针数组别名信息的表示融合在一起,更加精确地表示了指针别名信息,并在此基础上给出了指针数组过程内别名分析的基本框架与指

针数组别名信息的求取算法,从而把指针别名信息的分析从指针标量推广到指针数组,形成了一个过程内指针别名分析的完整体系.利用这个算法，能够同时有效而准确地求出指针数组与指针标量的别名信息,从而明显地提高C程序并行优化的效果.

* 本文研究得到国家自然科学基金、国家863高科技项目基金和国家教育部博士点基金资助。
** 如果没有特殊说明，文中皆假调整数占2字节，实数占4字节，32位寻址，即*sizeo*(*int**)=(*float**)=4

作者简介 黄波，1973年生，博士生，主要研究领域为并行编译。
臧斌宇，1965年生，副教授，主要研究领域为并行编译。
俞一峻，1972年生，博士，主要研究领域为并行编译。
朱传琪，1943年生，教授，博士生导师，主要研究领域为并行处理。
本文通读联系人：黄波，止海 200433，复旦大学并行处理研究所
作者单位：复旦大学并行处理研究所 上海 200433

参考文献

1 Stanford Compiler Group. The SUIF parallelizing compiler guide. Technique Report, Standford University, 1994
2 Blume B, Eigenmann R, Faigin K et al. The next generation in parallelizing compiler. In: Proceedings of the 7th Workshop on LCPC. 1994
3 朱传琪,臧斌宇,陈彤.程序自动并行化系统.软件学报,1996,7(3):180～186
(Zhu Chuan-qi, Zang Bin-yu, Chen Tong. An automatic parallelizer. Journal of Software, 1996,7(3):180～186)
4 Spillman T C. Exposing side effect in a PL/I optimizing compiler. In: Proceedings of the IFIP Conference. North-Holland, 1971. 376～381
5 Allen F E. Interprocedual data flow analysis. In: Proceedings of the IFIP Conference. North-Holland, Amsterdam, 1974. 398～402
6 Barth J M. A practical interprocedual data flow analysis algorithm. Communication of ACM, 1978,21(9):724～736
7 Aho A V, Sethi R, Ullman J D. Compilers Principles, Techniques and Tools. Reading, MA: Addison-Weslsy Publishing Company, 1986. 648～660
8 Wolfe M. High Performance Compilers for Parallel Computing. Reading, MA: Addison-Weslsy Publishing Company, 1995. 277～287
9 Wilson R P, Lam M S. Efficient context-sentive pointer analysis for C programs. In: Proceedings of ACM SIGPLAN '95 Conference on Programming Language Design and Implementation. La Jolla, CA, 1995. 1～12
10 Emami M, Ghiya R, Hendren L. Context-sensitive interprocedual points-to analysis in the presence of function pointers. In: Proceedings of ACM SIGPLAN'94 Conference on Programming Language Design and Implementation. 1994. 242～256
11 Liu Qiang, Zhang Zhao-qing, Ji Xiao-mei. Eliminating two kinds of data flow inaccuracy in the presence of pointer aliasing. In: Proceedings of the 1997 Conference on Advances in Parallel and Distributed Computing. Los Alamitors, CA: IEEE Computer Society Press, 1997. 410～415

本文1998-01-24收到原稿，1998-06-30收到修改稿