Malaysian Journal of Computer Science, Vol. 10 No. 2, December 1997, pp. 45-50

ALIASES AND THEIR EFFECT ON DATA DEPENDENCY ANALYSIS

R. Parimaladevi and R. K. Subramanian
School of Computer Sciences
Universiti Sains Malaysia
11800 Pulau Pinang
Fax: 04-6573335
Tel: 04-6577888 ext. 2128
email: rks@cs.usm.my

ABSTRACT

Parallelising compilers try to automatically convert sequential programs into parallel programs to be executed on the targeted parallel machine. The main task of the parallelising compiler is to locate the areas of potential parallelism in the sequential programs. The major problem in doing so is the data dependency in the programs. These could be identified by one or more passes of the program if unique variable names are used for the memory locations. If different variable names are used to point to the same memory location it causes a different dimension to the problem. If different variable names refer to the same location they are called aliases. Aliases could occur when there is a subprogram call. The parameters passed to the subprogram could be aliases in the subprogram itself or be aliases to the variable used in the calling program. Aliases occur during the usage of recursive data structures. Parameter scoping could also lead to alias problem when a global variable is passed to the subprogram as a parameter. To handle the problem of aliases the compiler will have to perform a detailed alias analysis so that suitable parallel codes could be gene-rated. The alias problem has been examined and methods of identifying the occurrences of aliases have been developed. The methods adopted by the tool for handling the aliases in C programs have been described.

Keywords: ***Parallel Programming, Dependency Analysis, Alias Analysis, Software Tools, Parallelising Compilers***

1.0 INTRODUCTION

Recent advances in technology has made parallel processing a reality. However, inadequate parallel software is a major factor in the very slow transition to parallel computing. Though there are many significant parallel applications and there are many parallel proces-sing hardwares built, system software for the new archi-tecture is still immature. VLSI provides for cheap and powerful microprocessors for assembly into an ensemble which has resulted in more unconventional and customized approaches to building parallel machines. The architecture of these parallel computers is totally different from Von-Newmann architecture. The existing software is written to suit the single-processor systems. Therefore these software need to be modified in order to be executed on parallel computers. But the programmers who are used to think and write programs in a sequential fashion find both developing parallel programs and modifying the existing sequential programs to parallel programs tedious. Besides, a large number of programs in sequential languages are still available and useful. Rewriting all of them to suit a parallel computer may be an avoidable tough task. An ideal way to use these bank of programs is to develop a tool which could convert the sequential programs to parallel programs either automatically or by interaction with the programmer. This tool is called parallelising compiler which would convert a sequential program into a parallel program that could run on parallel computers.

The major problem in parallelising a sequential program is the existence of data dependency in the program [1]. Data dependency not only occurs among the same variable names but also with their alias names which refer to the same memory location. Thus, the major obstacle for an accurate data dependency analysis is the existence of aliases in the programs.

Languages like Fortran and C, allow a memory location to be referred by more than one variable name, which causes aliasing in the program. Let, x and y are variable,
x is an alias of y, $x = y$ <=> iff, x and y are associated with same location during program execution [2].

There are two types of aliasing, static and dynamic[2]. Static aliasing results from explicitly specified storage overlays, such as EQUIVALENCE in Fortran. Dynamic aliasing is caused by the effect of the execution of certain statements such as pointer manipulation [3] or calls with reference parameters [4].

Let's consider the following statements:

```
S1:  x = 4 * k;
S2:  z = y;
```

In the absence of above information about alias, data dependency analysis indicates that statements S1 and S2 can be executed in parallel. But if y is an alias of x, there exists a flow dependency between these statements and parallelising them will lead to inconsistent result. Therefore, before we proceed to data dependency analysis, these alias names for a variable should be identified and the information should be passed to data dependency analyzer for an accurate analysis.

2.0 ALIASES

In C, static alias occurs when a pointer variable is assigned to an address of another variable [5]. Whereas, dynamic aliases occur during the manipulation of pointer data structures [3, 5, 6] and during the procedure calling [4].

2.1 Static Aliases

When a variable is declared as a pointer and assigned to the address of another variable, then we say that there exists aliases between the pointer variable and the second variable. Consider the following program code:

```
int  k;
int *j;

j = &k;
...
...
S1 :  *j = 10;
...
...
Sn :  k = 20;
```

Since *j* is referring to the address of k, **j = k*, they are aliases and, any changes made to **j* will also affect the value of *k*. Hence, in the above example, an output dependency between statement S1 and Sn, is hidden because of their alias names.

2.2 Dynamic Aliases

Dynamic aliases occur mainly under two situations:

i) alias during procedure calls and
ii) alias while using pointer data structures.

2.2.1 Aliases During Procedure Calls

C allows parameters passed to a subprogram in both pass-by-reference and pass-by-value method. However, the possibility of aliases is very high in pass-by-reference method, since they pass the address of the actual parameter to the subprogram. The program may pass same address to more than one formal parameter, and lead to the possibility of referring a variable with more than one name [7, 8]. Whereas, in pass-by-value method the value of actual parameter is copied to another location, thus any changes made to the formal parameter will not affect the value of actual parameter. When it comes to array and strings, although we use a pass-by-value method, to save the memory space, C handles it as pass-by-reference [9]. Such cases may lead to aliases.

Let us consider the program code below to see the situations where aliases could occur.

```
        void  calculate_1(int x, int y);
        void  calcualte_2(int *x, int *y);
        void  assign(int  k[], int l[]);

        int  G;

        main()
        {
                int  p,q,r,i,j;
                int Number[100];

                S1 : p =  10;
                S2 : q = 12;
                S3 : calculate_1(p,p);
                S4 : calculate_2(&p,&p);
                S5 : calculate_2(&G,&p);
                S6 : r =  p/100;
                S7
:calculate_2(&Number[i],&Number[j]);
                S8 : assign(Number,Number);
                return ;
        }

        void calculate_1(int  a,  int b)
        {
                int  s;

                S9 :  s = a + b;
                S10: a++;
                S11: b = G + a;
                return;
        }

        void  calculate_2(int  *a,  int *b)
        {
                S12: *a = G / 10;
                S13: G++;
                S14: *b++;
                return;
        }

        void  assign(int t[], int v[])
        {
                int i;

                for (i=0; i<100; i++)
                        S15: t[i] = v[i-1]   *  0.5;
                return;
        }
```

In statement S3, subprogram *calculate_1* is called by value. The value of *p* will be duplicated to *a* and *b* which will be stored in different memory locations. Thus *a <> b*, and there is no aliases in the subprogram *calculate_1*.

However, in statement S4, *p* is passed by reference to subprogram *calculate_2,* and causes same address to be passed to the both formal parameters **a* and **b*. Since **a* and **b* are referring to the same address, **a=*b*, these variables are aliases to each other [10]. This leads to output dependency between statements S12 and S14.

Aliases also occur, when a formal parameter is bound to an argument of a global variable [2, 7, 8]. This is obvious in statement S5, where the global variable *G* is passed-by-reference to subprogram. Hence, *G= *a*. Therefore, an output dependency exists between statements S12 and S13.

Consider statements S5 and S6, since in S5, the value of *p* may change (updated) in *calculate_2*, there exists a flow dependency between them [11].

In statement S7, **a = *b, *a* will be alias of **b,* iff *i = j* [10]. If this is the case, then an output dependency between statements S12 and S14 is hidden in the subprogram *calculate_2*.

In statement S8, although this is a pass-by-value method, since arrays are involved, now, both formal parameters are bound to the same address. Hence, *t = v*, and there exists a flow dependency in statement S15.

2.2.2 Aliases During Pointer Data Structures Manipulation

Dynamic data structures are defined by programmers. They could go as complicated as the programmers need. The well known data structures are link list and binary trees. The pointer fields defined in the data structure could have different meaning for each type of structure. For example, both binary tree and two way link list could have pointers left and right. For the link list right will point to the next node in forward direction, and left will point to the previous node in backward direction. However, for the binary tree the pointers go to the right and left direction. This means that each type of pointer data structure leads to aliases in their own way. Here let us trace how the two way link list and a binary tree cause aliases in C programs.

Consider the program code below, assume *head, r, s, t and u* are declared as dynamic data structures with two way link and *a, b and z* are variables.

```
S1 : r = head->right
S2 : s = r;
S3 : r->value = a;
S4 : b = s->value;
S5 : t = s->left;
S6 : r = r->right;
S7 : u = r->left;
S8 : u->value = z;
```

There is a flow dependency between statements S1 and S2. But, since this is dealing with pointers, *r* is alias of *head->right* and therefore, statement S2, can be rewritten as

s = head->right;

which will overcome the dependency that existed before the modification.

Now, consider statements S3 and S4, since *r* is an alias of *s*, there exists a flow dependency between them.

In statement S6, *r* moves to *next(right)* node, which leaves *r->left* to point to the previous node, which is *s;* This causes *s* to become an alias of *u* in statement S7. Thus an anti-dependency between statements S4 and S8 exists.

Now, consider the program code below, which deals with a leaf linked binary tree. Assume *p, q* and *s* are pointer data structures and *a* and *b* are variables.

```
S1 : r = root->right;
S2 : q = p->right;
S3 : s = p->left;
S4 : s = s->next;
S5 : s->value = a;
S6 : q-> value = b;
```

In statement S2, *q* points to the *right child* of *p;* whereas in statement S3, *s* points to the *left child* of *p*. In statement S4, *s* moves to the *next sibling,* which is *q*. (refer to the diagram below). Therefore, after statement S4, *s* and *q* are being aliases to each other. This causes an output dependency between statements S5 and S6.

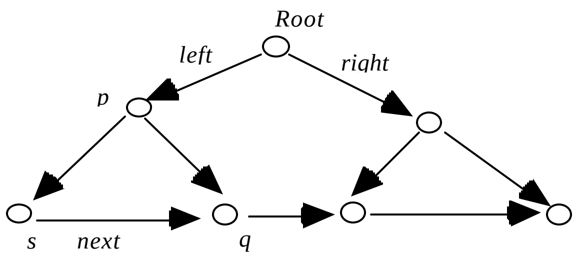

Fig. 1: A Leaf Linked

3.0 HANDLING ALIASES

Mohd. Saman [11] has discussed the Bernstein Sets method to collect information in the inter procedure analysis. This method is based on the set formation containing variables in programs, i.e., fetched and stored variable sets for each statement in the program. The paper also has discussed the parallelism of pass-by-value and pass-by-reference parameters which cause aliases in the called function. We have proposed a separate alias analysis to enhance the accuracy of data dependency analysis on the function calls. This work provides a more detailed analysis performed on the function calls to detect the occurrences of alias names. To handle the alias problem, the first thing we need to do is to identify and collect the alias names of each variable. Then this infor-mation is stored in a data structure, called alias table. This table will be maintained throughout the analysis of the sequential program. We replace the relevant variables in the program code with their alias names from the alias table. This will eliminate the spurious data dependencies and also make the real data dependencies visible. Hence, the data dependency analyzer will be able to come out with optimum and accurate result for parallelism.

3.1 Solution for Static Aliases

For the static aliases, (refer to the example given in 2.1), **j = k,* statement S1 have to be modified to:

S1: *k = 10;*

This will make the data dependency analyzer to detect a output dependency between statements S1 and Sn.

3.2 Solution for Dynamic Aliases

For passing two same addresses (pass-by-reference) and passing two same arrays (pass-by-value), the subprogram should be duplicated by passing the first parameter only. In the duplicated subprogram, the second variable name should be replaced by this first variable name. This will overcome the problem of hidden data dependencies. For example, consider the program code below:

the call : *calculate_2(&p,&p);*

the subprogram should be duplicated to:

```
void calculate_2dup(int *a)
{
        S12: *a = G / 10;
        S13: G++;
        S14: *a++;
        return;
}
```

and the call should be : *calculate_2dup(&p);*

Now it is obvious that there exists an output dependency between statements S12 and S13.

For passing global variables by address to a subprogram in a duplicated subprogram, that particular variable name should be replaced with the relevant global name. For example :

the call : *calculate_2(&G,&p);*

the subprogram should be duplicated to:

```
void calculate_2dup(int *b)
{
        S12: G = G / 10;
        S13: G++;
        S14: *b++;
        return;
}
```

and the call will be : *calculate_2dup(&p);*

Handling aliases in pointer data structures is a bit complicated, because each statement in the program should be analyzed and the best suited alias name should be replaced. Each statement could lead to different access path for each single pointer variable declared. Tracing the program code and then creating the access path corresponding to the main pointer (such as HEAD or ROOT) will enable us to identify the alias names for each pointer variable declared. Each type of dynamic data structure has its own axioms to be followed. This axioms will help us to trace the access path of each type of data structure.

A detailed work to handle the aliases caused by dynamic data structure has been proposed by Hummel et al. [6]. The dynamic data structure is handled by building an access path matrix for each statement, recording the "link" of the pointer variables. For instance, let us say there is a statement,

S1 : *p = q->next;*

The APM_{S1} will have entries of variables *p* and *q*, and in the column of *p*, the link *next* will be recorded. At this point, the axioms are not being applied on the links yet. Later, during the data dependency analysis, the APMs for those statements under analysis will be compared to detect the alias cases. Only at this point the axioms will be applied on the recorded links on the APMs.

We discuss a simpler way to handle the aliases caused by dynamic data structure. Instead of access path matrix for each statement in the program code, we propose an access path table which records all the pointer variables in the program. All the links will be recorded with respect to the main header or root of the structure. The axioms will be applied on the links and according to the outcoming result the link path of the respective pointer variable will be updated in the access path table. Hence, this access path table will contain the current position of the pointer variables along the dynamic data structure, with respect to the main header. At any one instance, by comparing the link path of the pointer variables the tool could identify the alias variables in the program.

Now, let us trace the program code below to identify the alias names for the pointer variables *head, r, s, t and u* in two way link list.

```
S1 : r = head->right;
S2 : s = r;
S3 : r->value = a;
S4 : b = s->value;
S5 : t = s->left;
S6 : r = r->right;
S7 : u = r->left;
S8 : u->value = z;
```

In the statement S1, *r = HR*

In statement S2, *s = r = HR*

Since *s is alias of r*, statements S3 and S4 contain a hidden flow dependency. To make it obvious, statement S3 should be rewritten to :

```
S3 :  s->value  =  a;
```

In statement S5, *t = sL*
= HRL,

according to the axioms of a two way link list : *p. LR = p, && p.RL = p;*

t becomes an alias of *head;* *t = H.*

In statement S6, *r = rR*
= HRR.

In statement S7, *u = rL*
= HRRL
= HR

Notice that at this point, *s = HR,* means that *u* is an alias of *s.*

In statement S8, *u->value* is being updated causing an anti-dependency between statements S4 and S8. This hidden dependency will be obvious when statement S8 is rewritten as :

```
S8 :  s->value  = z;
```

Thus, after modification, the above program code will look like:

```
S1 :  r = head->right;
S2 :  s =  head->right;
S3 :  s->value = a;
S4 :  b =  s->value;
S5 :  t  =  s->left;
S6 :  r = r->right;
S7 :  u = r->left;
S8 :  s->value =  z;
```

The difficult part in doing the modification is to choose the best suited variable to be replaced when each alias names being encountered. For example, when considering statements S3 and S4, if we modify statement S4 instead of statement S3, as

```
S4 :  b =  r->value;
```

although the flow dependency between them are clear, this will hide the anti-dependency between statements S8 and S4 since *u* is an alias of *s*, not to *r* at this point, since *r* is moved to another location in statement S6.

Now, let us trace the alias properties in a leaf linked binary tree through the program code below; assume *r, q and s* are pointer data structures and *a and b* are variables.

```
S1 :  r = root->right;
S2 :  q = p->right;
S3 :  s = p->left;
S4 :  s =  s->next;
S5 :  s->value = a;
S6 :  q-> value = b;
```

In statement S1, *r = RoR*

In statement S2, *q = pR*
= RoRR

and in statement S3, *s = pL*
= RoRL

In statement S4, *s = sN*
= RoRLN

These information for each pointer variable will be recorded in *access-path-table.* The comparison of the access paths for each pointer variable together with the axioms will give us the alias properties for a leaf linked binary tree.

Axioms for a leaf linked binary trees are:

A1: f.LN = f.R
A2: f.RLN = f.RR
A3: f.LRN = f.RL

and axiom A3 can be expanded to:

f.LRxN = f.RLx, where x is the counter for R and L.

Consider the pointer variables *q = RoLR;*
and s = RoLLN;

For the above case assume *f* as *RoL* which is the property of both pointer variable;
Now, we have:

s = f.LN; q = f.R;

which is true according to axiom A1. Therefore, we have found that *s* is an alias of *q* after statement S4.

Thus, the program code should be changed to,

```
S1 :  r = root->right;
S2 :  q = p->right;
S3 :  s = p->left;
S4 :  s =  s->next;
S5 :  s->value = a;
S6 :  s-> value = b;
```

Now, the output dependency between statements S5 and S6 became obvious for the data dependency analyzer.

Aliasing variable names cause hidden data dependencies between statements. So for an accurate data dependency analysis aliases should be identified and proper actions should be taken. This will enable parallelising a sequential program with very high efficiency.

ACKNOWLEDGMENTS

The grants provided for the above research work by University Sains Malaysia and IRPA program are gratefully acknowledged.

REFERENCES

[1] R. K. Subramanian, "Reusing Sequential Programs on Parallel Platforms", *Proceedings of the IASTED International Conference on Modeling and Simulation held in Pittsburgh, USA*, April 1995.

[2] Hans Zima, Barbara Chapman, "*Supercompilers for Parallel and Vector Computers*", Addison-Wesley Publishing Company, 1991, pp. 95-109.

[3] Joseph Hummel, Laurie J. Hendren, Alexandru Nicolau, "A General Data Dependence Test for Dynamic, Pointer Based Data Structures", In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994, pp. 218-229.

[4] J. P. Banning, "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables", *6th ACM Symposium on Principles of Programming Languages*, 1979, pp. 29-41.

[5] Joseph Hummel, Laurie J. Hendren, Alexandru Nicolau, "A Framework for Data Dependence Testing in the Presence of Pointers", *Proceedings of the 23rd Annual International Conference on Parallel Processing*, August 1994, pp. 216-224.

[6] Joseph Hummel, Laurie J. Hendren, Alexandru Nicolau, "Path Collection and Dependence Testing in the Presence of Dynamic, Pointer Based Data Structures", *Proceedings of 3rd Workshop on Languages, Compilers and Run-time Systems for Scaleable Computers*, May 1995, pp. 15-27.

[7] Alfred V. Aho, Jeffrey D. Ullman, "*Principles of Compiler Design*", Addison-Wesley Publishing Company, 1978, p. 506.

[8] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, "*Compilers-Principles, Techniques and Tools*", Addison-Wesley Publishing Company, 1988, p. 648.

[9] Beatrice Creusillet, Francois Irigoin, "Interprocedural Array Region Analyses", In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[10] Robert W. Sebesta, "*Concepts of Programming Languages*", The Benjamin/Cummings Publishing Company Inc., 1993, pp. 131-136.

[11] M. Y. Mohd. Saman, D. J. Evans, "Inter-procedural Analysis for Parallel Computing", *Parallel Computing 21*, 1995, pp. 315-338.

BIOGRAPHY

R. K. Subramanian is currently a Professor with the School of Computer Sciences, Universiti Sains Malaysia. He has over 30 years of teaching and research experience. His research interests include Parallel Processing, Artificial Intelligence and Computer Networking. He received his B.E. (Hons) in Electrical Engineering from Annamalai University and M.Sc (Engg) in Electrical Engineering from University of Madras and Ph.D. in Computer Science from the Indian Institute of Technology, Delhi. He has around 60 publications to his credit.

Parimaladevi graduated from Universiti Malaya, in 1994, with a B.Sc.Comp (Hons) degree. After teaching for a short period in a private institute she has taken up research. She is currently working for her masters degree in the field of Parallel Processing, in the School of Computer Sciences, Universiti Sains Malaysia. The research focuses on developing a parallelising compiler for C programs. Special attention is given to the area of inter-procedural analysis and the usage of dynamic data structures in C programs.