Data Dependence Analysis In Presence Of Inheritance and Polymorphism*

Li Xiaoming, Chen Daoxu and Xie Li
State Key Laboratory of Novel Software Technology
Department of Computer Science, Nanjing University, Nanjing 210093
Email: xmli@dislab.nju.edu.cn, cdx@nju.edu.cn

Abstract:

The data dependence analysis is a hard problem, particularly in the presence of data structures similar to the pointer. The inheritance and the polymorphism in the object-oriented language provide the program designing and the software engineering with new methods. But at the same time, they bring about more barriers in the dependence analysis for object-oriented language. This paper proposes an object-oriented data dependence analysis model —ODAM. The model can present and analyze the specific concepts in OO languages. We mainly discussed the framework and the key techniques of ODAM, including the object hierarchy graph, the merge of the read-write set and the dependence analysis based on the object hierarchy graph.

Keyword:

Dependence analysis, Object Hierarchy Graph, Parallelism, Inheritance, Polymorphism

1. Introduction:

The most important aspect of the automatic parallelizing compile is the data dependence analysis on programs. When compared with the procedural languages, many new concepts are introduced in OO languages. Among them, the following three have the most significant effects on the dependence analysis. They are the object, the inheritance and the dynamic binding. We discuss the three aspects respectively in the following parts.

The object is the basic idea in describing data structures in OO languages. The object and the variable are two different concepts in these languages, which is different from that in the procedural language. According to the declare type, variables can be divided into 2 categories, one is the simple variable, and the other is the object variable. The type of the simple variables is the primitive type of a language, and what are stored in them are values of the respective types. The declare type of the object variables is a class, and they can be viewed as special pointers. What the object variable stores isn't the object, but the binding relationship with the object. Only the binding relations of the object variable with objects are changed when assigning a new value to them. The object variables have some common properties with the normal pointers, such as two pointers can be alias to each other. When considering the OO characteristics, the object variable has more unique properties, such as the inheritance relation among objects according to the class hierarchy graph.

The inheritance describes the hierarchical relation between the data structures (class) in the OO language. Because of this, the object variables in OO languages and

* This work is supported by the National Climb Plan B and the 863 High Technology Research & Development Project.

0-7695-0589-2/00 $10.00 © 2000 IEEE

the pointer in procedural languages show many differences. The object variable can bind not only the object of its declared type, but also the object of the subclass. On the contrary, the pointer can only point to the memory units whose type is the pointer's declared type. We can conclude from the above that when analyzing the data dependence in the OO language, there are data dependences not only between two object variables of the same type, but also between the two who have the inheritance relations.

The dynamic binding makes it impossible to get all the information of the call at compile time. And consequently, it's more difficult to do the interprocedural analysis. An object variable can bind objects of different types, so if the method is overridden, the compiler cannot know which method is to be called at runtime. On the contrary, in languages like Fortran, the precise call graph can be got at the compile time, which make the accurate interprocedural analysis possible. So in the analysis on the OO language, some conservative algorithms must be used to do the interprocedural analysis, and it must include all possible methods. It can be seen that in parallelizing the OO language, the efficiency and the accuracy of the conservative algorithms affect the mining of the parallelism greatly.

Within the context, we propose the Object Dependence Analyze Model (ODAM) and implement the ODAM based on Java. The main idea of ODAM is to include the class hierarchy and the virtual methods into the internal format of the dependence analysis, which make the analyze algorithm can access these information fully. So, The model can handle the peculiar characteristics in OO language, including object, inheritance, dynamic binding, etc.

The rest of this paper is organized as following: Section 2 discusses the ODAM in detail. Section 3 gives some examples, Section 4 gives some related works and Section 5 is the conclusion.

2. ODAM

ODAM mainly deals with the holdbacks brought by the OO concepts to the parallelism mining in an object-oriented way. The key idea is generating the abstract object according to the class hierarchy. Based on that, it gives the rules to test the dependence between the abstract objects and the real ones, and it tries the best to get the most precise dependence relation possible. The following part describe the ODAM from several aspects: Section 2.1 clarifies the problems ODAM solves; Section 2.2 introduces the concept of the Object Hierarchy Graph; Section 2.3 gives the rules how to calculate the define set and the use set including abstract objects; Section 2.4 gives the rules of dependence testing and Section 2.5 discusses the implementation of ODAM.

2.1 Problem Clarification

The problems in the data dependence analysis can be divided into 2 largely-distinct categories. The first is the problem concerns only the dependence among object variables (Figure 1). The second involves analyzing the dependence between two fields of objects (Figure 2).

We shall refer the first problem as Variable Dependence Problem (VDP), and the second as Field Dependence Problem (FDP). The essence of VDP is to analyze the binding relation between the object variables and the real objects. Then naturally, the analysis techniques for the scalar variables in procedural language can be applied in VDP. On the other hand, the key point in FDP is to test whether the objects bound by the object variables are equal. The counterpart in the procedural language is to test whether two pointer point to the same memory location. But in FDP, much more factors must be considered, including inheritance, encapsulation, etc. Also, FDP must be handled in the interprocedural analysis in OO languages, because methods belong to class and they are dynamic bound. This paper mainly concerns with FDP.

2.2 Object Hierarchy Graph

The inheritance relation must be dealt with when considering FDP. The class hierarchy graph is usually used when depicting the inheritance relation among classes. That gives us implication that we can also use the

```
...
ClassA obj1,obj2;
S: obj1 = new ClassA();
T: obj2 = new ClassA();
...
U: obj1 = obj2;
```

There exists output dependence
between S and T, and flow
dependence between T,U

Figure 1:Variable Dependence Problem

```
...
ClassA obj1;
S:   obj1 = new ClassA();
T:   obj1.field1 = ...;
...
U:   ... = obj1.field1;
```

There exists flow dependence between
S and T, but whethert here is dependence between
T and U depends on if obj1 is
redefined in statements between T,U

Figure 2:Field Dependence Problem

hierarchy graph when dealing with the inheritance relation among objects. We can apply the hierarchical relations to test if two object variables may bind the same object. First, the concept of real object is introduced:

Definition 1 Real Object(RO): RO is a data structure, which identifies uniquely the real objects in program generated with the operator new. The object type is also recorded in RO.

In Figure 3, we can distinguish the different objects bound with the object variable obj in different phases of runtime with the use of RO1 and RO2. If not stated explicitly, the following part use Java as the example language.

```
Assume ClassB is a direct subclass of ClassA
    ClassA obj;
    obj = new ClassA ();
    //This object is identified as RO1;
    ......
    obj = new ClassB ();
    //RO2;
```

Figure 3:Use of RO

We need to say some more words if the object is generated in a loop. There are many ways to deal with this. For example, we can assign the same RO to all the objects generated in the loop while ignoring the infection of the iteration. This is a method with low precision. On the other extreme, different ROs will be assigned to objects generated in different iteration, which can improve the precision of the analysis. We can see from this that the concept RO has good adaptability that it can reach arbitrary precision the analysis requires.

Definition 2 Abstract Object (AO): Abstract object is a virtual object. Each class and array type corresponding to one unique abstract object. The AO represents all the instances in the type. The abstract object corresponds to type ClassA is showed as AO_{ClassA}.

The abstract object treats the objects of the same type as a whole. The define and use of a field of a abstract object is viewed as the define and use of the corresponding field of all the object of the class.

Definition 3 Let CLASS denotes all the classes in the program. The function type: $AO \cup RO \rightarrow CLASS$ is defined as following: type(obj) = the type of obj, $obj \in AO \cup RO$
The relation $super \subseteq CLASS \times CLASS$ is definded as following: $(ClassA,ClassB) \in super$ if and only if ClassA is the direct superclass of ClassB. Consequently, super* and super+ can be definded.

For example, type(AO_{ClassA}) = ClassA. As to the RO1 and RO2 in Figure 3, type(RO1) = ClassA, type(RO2) = ClassB and $(ClassA,ClassB) \in super$.

Based on the above, we can give the definition of the Object Hierarchy Graph, whose purpose is to show the

inheritance relation among objects, including both real objects and abstract objects.

Definition 4 Object Hierarchy Graph: It has two component, which can be wrote as (V,E). $V=AO \cup RO$ is the set of vertex. E is the set of directed edges. $(v1,v2) \in E$ if and only if one of the following is satisfied: (1) $v1,v2 \in AO$, $(type(v1),type(v2)) \in super$; (2) $v1 \in AO, v2 \in RO$, $type(v1)=type(v2)$.

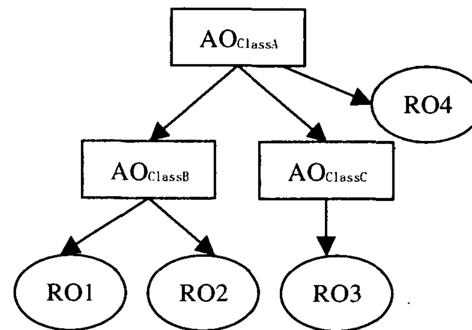
From the definition of OHG, it can be proved that the OHG is a directed acyclic graph(DAG) if no cycles present in the class inheritance graph (The cycles of inheritance are forbidden in most compilers). What the OHG defined is the containing relation among objects being read or wrote. If a node in the OHG is accessed, it will be dealt with as all the node in the connected subgraph containing the origin node have been been read or wrote. In Figure 4, (a) shows the classes' inheritance relations; (b) is a program fragment and (c) gives the OHG corresponding to (a) and (b) according to the definition 4.

```
class ClassA{
...
}

class ClassB extends ClassA{
...
}

class ClassC extends ClassA{
...
}
```

(a) The classes

```
ClassA obja;
ClassB objb;
ClassC objc;

obja = new ClassB(); //RO1;
objb = new ClassB(); //RO2;
objc = new ClassC(); //RO3;
obja = new ClassA(); //RO4;
```

(b) The program fragment



(c) OHG

Figure 4 The Object Hierarchy Graph

2.3 Calculate Define Set And Use Set

According to the definition of VDP and FDP, the statements without method call can be divided into 2 categories. (1) Simple statement: Only the simple type variable and the object variable are definded and used; (2) Field statement: Some fields of objects are definded and used. In practice, a statement may be both the simple statement and the field statement. But in order to improve the precision and the efficiency of analysis, the statement can be converted into one of them in the front-end. The detail of the converting techniques won't be discussed here.

We use SIMPLEVAR to denote the set of all simple variable, and OBJVAR to the set of object Variable. The set of all variable is $VAR=SIMPLEVAR \cup OBJVAR$. A special set OBJFIELD is introduced to represent the set of all fields of objects (RO or AO).Then all the memory units needed in the analysis form a set $MEM=VAR \cup OBJFIELD$.

Definition 5 Given $x \in OBJFIELD$,two functions can be definded on x: Obj: $OBJFIELD \rightarrow AO \cup RO$. Obj(x) return the object of x; Fld: $OBJFIELD \rightarrow string$. Fld(x) return the name of the field of x. Because all fields have

```
...
ClassA obj;
S1: obj = new ClassA();         //RO1
S2: ... = obj.field1;
...
S3: obj.field1 = obj.field2 + 1;
//Assuming the algorithm has lose
//the track of obj
```

def(S1)={obj}
use(S2)={RO1.field1}
def(S3)={AO_{ClassA}.field1}
use(S3)={AO_{ClassA}.field2}

Figure 5:Calculate the def/use set

unique names in a class, two fields can be distinguished by the names.

Definition 6 def: statement $\rightarrow 2^{MEM}$ give the define set of a statement;
use: statement $\rightarrow 2^{MEM}$ give the use set of a statement;
bind: OBJVAR $\rightarrow$ 2(AO $\cup$ RO) give the set of object bound to a object variable.

The def/use set of a simple statement can be got directly. The variables appearing in the left side of the assign symbol are added to the def set, and the variables in the expressions are added to the use set.

As to getting the def/use set of a field statement, the object binding set must be got first for all the object variables in the statement. In order to ensure the correctness of the analysis, all the corresponding field in the object binding set should be added to the def/use set respectively. The common method to calculate the object binding set is to track the binding relation of the object variable following the program flow. If the tracking algorithm is precise enough, it can accurately determine the object binding set of each object variables in the statement, i.e. bind(V) $\subseteq$ RO. On the other hand, the track may be lost when dealing with the loop or the method call statement. If no assignment to the object variable occur after losing the track, it can be conservatively estimated that the object bound to the variable is the abstract object of the declared type of it. From the definition of abstract, we can prove that the conservative assumption is correct. The example of how to calculate the def/use set of statements is shown in Figure 5.

There are fundamental differences between the calculation of the def/use set for the simple statement and for the call-method statement. Because of the polymorphism introduced by the OO language, a call-method statement corresponds to more than one possibilities of call at runtime. We employ the conservative approach that the def/use set are calculated as the union of the corresoponding sets of all the method possibly called. Considering the definition of the dependence relation (Definition 8), only the field statement and the call-method statement have effects on the def/use set at the call site. It's the same as the analysis of the field statement, the object binding set of the object variables used in the method, including the call object, the object variables as the arguments, must be got first. The calculation of the def/use set can be more precise with the improvement of the analyze algorithm without any bound, if the def/use set is expressed in OHG. For example, two extremes are of special interest: (1) Estimate the data dependence without analyzing the method at all. In this condition, if we want to calculate the def/use set, first we must get the corresponding abstract of the object variables in the method called; second, the def/use set of the method can be set as the union of all the fields of the objects in the connected subgraph of OHG containing the corresponding node of the abstract object. Obviously, the def/use set got in this way is the superset of the true def/use set of the method. So the correctness is guaranteed. (2) In some simple conditions, the object binding set can be got precisely. So the techniques such as the inline analyze can be used to get all the def/use set of the statements in the method, with the same idea of tracking. Consequently, the def/use set of call statement is the union of all these sets. The example of calculating the def/use set of the call method statement is shown in Figure 6.

```
...
Parent obj;
...
S1: obj.func();
//Given that bind(obj)={RO1,RO2} and
//type(RO1) = Parent type(RO2) = Child
...
S2: obj.func();        //bind(obj) = {AO_Parent}
...
```

The method func is overridden in subclass Child

So:

$def(S1)=def(RO1.func) \cup def(RO2.func);$

$def(S2)=def(AO_{parent}.func) \cup def(AO_{Child}.func);$

$use(S1)=use(RO1.func) \cup use(RO2.func);$

$use(S2)=use(AO_{parent}.func) \cup use(AO_{Child}.func);$

Figure 6:Calculate the def/use set of call-method statement

2.4 Dependence Analysis

We can do the dependence testing after get the def/use set of all the statement. The concept "layer" is introduced to describe the range of the dependence test. The data dependence relation is only valid between two nodes in the same layer. The inter-layer dependence is meaningless.

Because the dependence between the nodes in the loop and that out of the loop don't have explicit meaning, and the same is true to that between the nodes in a method and the nodes in the same method of the call site, we can give the condition to set up a new layer: (1) When analyzing a loop statement; (2) When analyzing a call statement. The loop statement and the method-call statement can viewed as a virtual node when doing the dependence testing between them and the other nodes in the same method. The def/use set of a virtual node is the union of the def/use sets of all the nodes in its sublayer.

Three categories of dependence is considered in most cases. They are the Flow Dependence, the Anti-Dependence and the Output Dependence. When not considering the OHG, they can be defined as: Given two node S1 and S2 in the same layer, and S2 is accessable from S1 along the control flow of the program:

(1) Flow Dependence (δ_f):

$$S2\,\delta_f\,S1 \leftrightarrow def(S1) \cap use(S2) \neq \varnothing$$

(2) Output Dependence (δ_o):

$$S2\,\delta_o\,S1 \leftrightarrow def(S1) \cap def(S2) \neq \varnothing$$

(3) Anti Dependence (δ_a):

$$S2\,\delta_a\,S1 \leftrightarrow def(S1) \cap def(S2) \neq \varnothing$$

When the OHG is introduced in the dependence testing, the testing conditions above aren't valid any more, because there may be abstract object in the def/use set. The abstract object should be looked upon as all the objects of the class, so we cannot determine the dependence relation only by comparing two memory units in the def/use set are equal. The definition of the intersection of two memory unit sets should be extended to test the dependence of two def/use set based on OHG.

Definition 8 Extended Containing Relation $\in_e$: Given $x \in MEM$, $S \subseteq MEM$, $x \in_e S \leftrightarrow$

$\exists y : y \in S \wedge (((x \in VAR) \wedge (x = y)) \vee ((x \in OBJFIELD) \wedge (y \in OBJFIELD) \wedge$ (obj(x) is in the connected subgraph of OHG that containing obj (y)) $\wedge$ (fld (x) = fld (y))));

Extended Intersection Operator $\cap_e$: Given M1, $M2 \subseteq MEM$, $x \in MEM$, $M1 \cap_e M2$ is defined as:

$$M1 \cap_e M2 = \{ (x \in M1 \wedge x \in_e M2) \vee (x \in_e M1 \wedge x \in M2) \}$$

Based on the definition of the extended Intersection operator, we can define the extended dependence testing conditions which can deal with the abstract object: Given two node S1 and S2 in the same layer, and S2 is accessable from S1 along the control flow of the program:

(1) Extended Flow Dependence (Δ_f):

$S2 \Delta_f S1 \leftrightarrow def(S1) \cap_e use(S2) \neq \varnothing$

(2) Extended Output Dependence (Δ_o):

$S2 \Delta_o S1 \leftrightarrow def(S1) \cap_e def(S2) \neq \varnothing$

(3) Extended Anti Dependence (Δ_a):

$S2 \Delta_a S1 \leftrightarrow def(S1) \cap_e def(S2) \neq \varnothing$

2.5 Implementation and Application

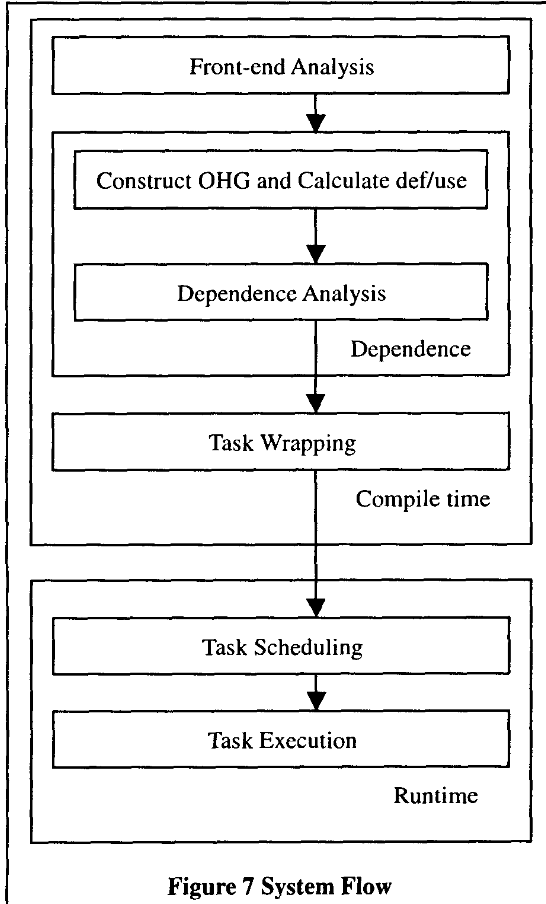


Figure 7 System Flow

We implement ODAM in JAPS [9] based on Java language. Two main aspects should be noted: one is the use of ODAM in compile time; the other is the support to ODAM at runtime. Figure 7 is the main flow of the system.

In front-end analysis, we get the class hierarchy graph the primitive information about the variables being wrote or read. In the second phase, the OHG is constructed according to the definitions 1 ~ 4, and the def/use set is calculated after the definitions 5~7. Both the steps are in term of the request of the dependence-analyzing algorithm. In the phase of dependence analysis, definitions 8 and the concept layer are used to calculate the dependence. Different algorithms can be used and they may be replaced by more precise algorithm with the advance of the research. ODAM provide sufficient support to these.

In the following 3 phases, how to support the abstract object must be considered. The abstract object in ODAM is a virtual concept introduced with the purpose of conducting the data dependence analysis. There aren't any abstract objects in real programs. And different supports are needed in the shared-memory system and the distributed-memory system:

(1) Shared-Memory System

In Shared-Memory Systems, if given two tasks T1 and T2 and the data conflicting set between T1 and T2 contains abstract objects, T2 must be executed after T1. When T1 has finished, all the objects are wrote back to the shared memory. The system states are synchronized. The abstract object has no effect on the execution of T2. So in shared-memory system, it's necessary only to consider the abstract object concept in the analyzing phase. No runtime support is needed.

(2) Distributed-Memory System

In Distributed-Memory System, if the abstract object AO_{ClassA} is contained in the data conflicting set between T2 and T1, T2 must be executed after T1. When T1 has finished, some objects in the conflicting set must be transferred from T1 to T2, according the data dependence types ($\Delta_f, \Delta_o, \Delta_a$). If AO_{ClassA} is needed to be transferred, because it doesn't exist at all, all the real objects (RO) in the connected subgraph that containing AO_{ClassA} should be transferred to T2 to synchronize the system. So as to the distributed-memory system, it is needed to determine the

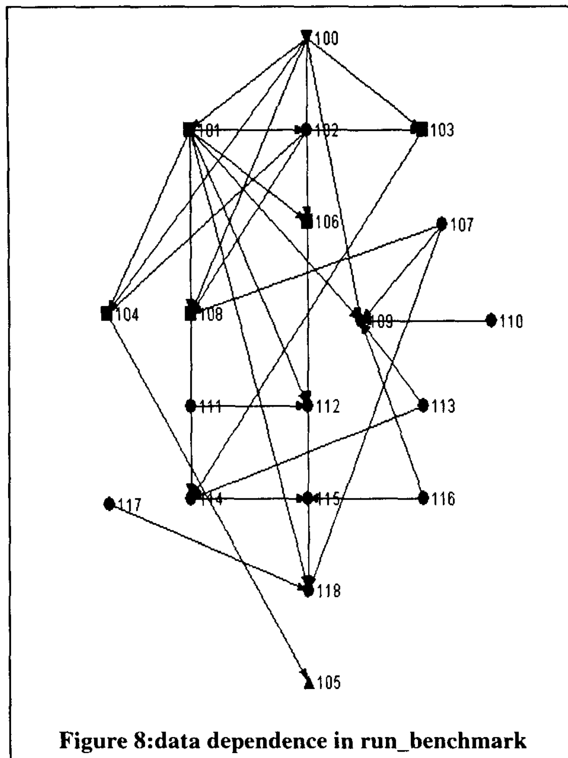


Figure 8:data dependence in run_benchmark

connected relation in OHG at runtime.

3. Case Study

The program to be analyzed is the Java version of Linpack (the source code can be downloaded from http://www.cs.cmu.edu/~jch/java/linpackloop.java). We analyzed the key method run_benchmark() in Linpack with ODAM. The interprocedural analysis is applied in the process. The dependence graph got automatically is shown in Figure 8. In Figure 8, the triangle symbols represents the head and the tail node; the round symbols represent simple node; and the rectangle symbols represent method-call node.The directed edges represent the data dependence relation between nodes. If the label of the head less than that of the tail of the edge, the dependence is Δf, else it is Δa or Δo.It ought to be notified that Figure 8 is for the purpose of showing the data dependence fully, so we don't include the granularity control.

4. Related Work

There exist many parallelizing compile technologies for the OO languages. The emphasis of these technologies is mostly on the pointer-like data structures and the interprocedural analysis. The main idea of the approach from J.Hummel et.al.[2] is to label data structures with path expressions. The dependence tester tries to prove that dependence between two paths is impossible based on a group of axioms. If the proof exists, no dependence is possible. The advantage of the approach is that it has a well-formed theory basis. But it's impractical because the axioms can't be given easily. W.Amme et.al [3] proposed a pointer analysis model based on the A/D graph. It has a similar concept to the abstract object in ODAM, which is the abstract pointer. The limitation is that the abstract pointer can't present the object hierarchy in OO language directly. In order to solve the alias set problem, A. Deutsch [4] develop a different memoryless method, which employing only algebraic methods. Due to the lack of the support to the update of the binding relations, this method has no way to calculate the must-alias information. So it can only tell if two pointers conflict, but can't give the data set the pointers depending on. M. Sagiv [5] solved the dependence problem on the cyclic data structure by analyzing the possible shapes of data structures. D.Bairagi [6] and D.Grove[7] developed approaches to construct the precise call graph of programs in OO languages. The Javar project [8] from Indiana University mainly exploits the parallelism in the loop and recursive methods in Java program. It has not only compile-time analysis, but also runtime dependence test. Because the emphasis is on the latter, it doesn't exploit satisfactory parallelism from the analysis.

5. Conclusion

ODAM can deal with the OO characteristics such as inheritance, encapsulation and dynamic binding in the data dependence analysis in a convenient way and with high efficiency. It provides supports to both the parallelism mining and the parallel task executing. ODAM can be applied to all the popular OO languages, for it provides the further research on the data dependence analysis with a consistent internal format. We will put our emphasis on

integrating ODAM with other techniques in the future, which we believe will exploit the parallelism of programs more fully.

References

[1] James R.Larus and Paul N.Hilfinger,"Detection Conflicts between Structure Acesses", *Proceedings of the SIGPLAN'88 Conferences on Programming Language Design and Implementations*,June 1988.

[2] Joseph Hummel,Laurie J.Hendren and Alexandru Nicolau,"A General Data Dependence Test for Dynamic Pointer-Based Data Structure",*Technical Report*, U. of California,Irvine,1994

[3] Wolfram Amme,Eberhard Zehendner,Data dependence analysis in programs with pointers, *Parallel Computing* ,1998, 24, 505-525

[4] A. Deutsch, Interprocedural may-alias analysis for pointers:beyond k-limiting, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Languages Design and Implementation*,Orlando, 1994, 230-241

[5] M. Sagiv, T.Reps, R.Wilhelm, Solving shape-analysis problems in languages with destructive updating, *SIGPLAN-SIMACT Symposium on Principles of Programming Languages*,FL, 1996, 16-31

[6] D. Bairagi, S.Kumar, D.P. Agrawal, Precise Call Graph Constrction for OO Programs in the Presence of Virtual Functions, IEEE Transactions on Parallel and Distributed Systems, 1997, 8(4),412-416

[7] D.Grove, G. Defouw, J.Dean, et.al. Call Graph Construction in Object Oriented lanaguage, *OOPSLA'97 Conference Proceedings*, Atlanta, GA, October, 1997

[8] Aart J.C.Bik and Dennis B.Gannon,"Automatically Exploiting Implicit Parallelism in Java", *Technical Report TR473*, Indiana University,Jan. 1997.

[9] Du Jiancheng, Chen Daoxu and Xie Li: "JAPS:An Automatic Parallelizing System Based on JAVA", *Science in China*, 1999 ,vol 3, 279-288