Demand-driven Alias Analysis Implementation Based on Open64

Xiaomi An
(annyur@gmail.com)

Abstract

In this paper, an implementation of a demand-driven alias analysis [7] in Open64 is presented. In the algorithm, a program expression graph is constructed based on all the expressions and assignments in the program, and the memory alias problem is formulated as a CFL-reachability problem. To deal with field accesses of structs which are common in multi-media applications, a field-sensitive extension of the original algorithm is also implemented. Currently, the field-sensitivity assumes ansi-compliant programs.

By evaluating the implementation using some spec2000 programs, we found that the scalability of the original algorithm is not able to give enough precision in a reasonable compile time. To improve the scalability, a new one-level flow demand driven algorithm is developed, in which the hierarchical state machine is simplified by making machine M transitive while still keeping machine V non-transitive. The algorithm can achieve the same precision as Das's algorithm [3], but uses a demand driven approach.

Although the new one-level flow algorithm lost some of the precision for making machine M transitive, it can finish analysis of more queries for a given reasonable compile time and give more definite alias results. In other words, it can give better precision than the original algorithm and is more practical for product compiler. From experiment results, we can see that the algorithm can finish about half of the queries in a short compile time, and give much more "not aliased" results than the original algorithm.

1. Introduction

The set-union based alias classification [1] in Open64 has been used for many years, and played an important role in memory disambiguation as well as SSA based IR building in the global optimizer. The precision of alias analysis directly determine the effect of all the optimizations based on it, such as partial redundancy, instruction scheduling, etc. Since then more program analysis techniques have been developed, such as software security static check, data race check for multi-threaded application. The effectiveness of these analysis techniques all heavily rely on the precision of alias analysis.

Set-inclusion based point-to analysis algorithm was developed by Andersen in 1994[2], known for its better precision with $O(N^3)$ complexity. This approach has been used by many modern compilers to improve the precision of their alias analysis. At the same time, plenty of new ideas have been developed and published to improve the scalability of the set-inclusion based algorithm.

These new methods tried to solve the scalability issue from different aspects, and can be summarized into the following three kinds:

1) Partial set-inclusion approach: Das presented a low cost algorithm which lies between Steenguaard's algorithm and Andersen's algorithm [3]. The algorithm uses a restricted form of subtyping to avoid unification of symbols at the top levels of pointer chains in the points-to graph, while using unification elsewhere in the graph.

2) Constraint graph simplification approach: Manuel showed that online elimination of cyclic constraints can make implementation more

scalable [4]. Similarly, Atanas Rountev developed an offline variable substitution algorithm which reduced the cost of Andersen's points-to analysis substantially [5].

3) Demand-driven approach: Some researchers noticed that some of the efforts spend in points-to or alias analysis are not effective, and developed demand-driven approaches to do points-to analysis or alias analysis just on demand. For example, Heintz and Tardieu redesigned Andersen's deduction rules by adding conditions to the rules, so that only necessary analysis is derived for querying [6]. Xin Zheng and Rugina presented a demand-driven alias analysis in 2008 [7]. In their algorithm, the computation of alias queries was formulated as CFL(context free language)-reachability over graph representation of assignments and pointer dereference relations, and they claimed that the demand-driven alias analysis will do less work than demand-driven points-to analysis.

In this paper, we try to enhance the alias analysis of Open64 by incorporating a demand-driven alias analysis which is based on Andersen's set-inclusion based algorithm. To further improve the scalability, we simplified the hierarchical state machine and developed a one-level flow demand-driven algorithm.

The reason we choose a demand driven algorithm is in three folds: 1) it need not calculate all the alias relationship at a time, but do necessary analysis on demand; 2) It needs less memory space since it need not save the alias data for the whole program, some of which may not be used at all. 3) In case only partial information is available, demand driven algorithm can recover the needed info naturally. Currently, the implementation is used in intra-procedural analysis, and we hope to extend and incorporate it into inter-procedural analysis in the future.

2. Demand-driven alias analysis via CFL_Reachability

In this section, we give a brief introduction of how the memory alias problem can be formalized into a CFL-Reachability problem [7].

The C-like program which manipulates pointers can be represented by a *Program Expression Graph* (PEG). The PEG is a graph representation of all expressions and assignments (both explicitly and implicitly) in the program. The nodes of the graph represent program expressions (including symbols) and the edges are of two kinds:

i) Dereference edges (D): for each dereference *e, there is a D-edge from e to *e, and at the same time, there is a $\overline{D}$-edge from *e to e.
ii) Assignment edges (A): for each assignment e1 = e2, there is a A-edge from e2 to e1, and at the same time, an $\overline{A}$-edge from e1 to e2.

Unlike traditional alias analysis algorithm, two kinds of aliases are defined here:

i) Memory aliases: two dereference expressions are memory aliased if they might access the same memory location, i.e. their addresses are value aliased.
ii) Value aliases: two expressions are value aliased if they might evaluate to the same pointer value.

The may-alias problem can be formulated with a context-free grammar shown as the following. EBNF notation is used to represent the grammar, where "?" indicates an optional term, "*" is the Kleene star operator. Symbol D, $\overline{D}$, A, $\overline{A}$ are terminals and denote D-edge, $\overline{D}$-edge, A-edge and $\overline{A}$-edge separately. F and $\overline{F}$ are non-terminals, denoting value flows.

Memory aliases: $M ::= \overline{D}VD$

Value aliases: $V ::= \overline{F}M?F$

Flows of values: $F ::= (AM?)*$,

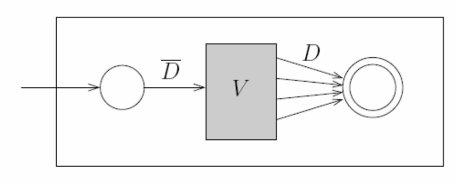
$$\overline{F} ::= (M?\overline{A})*$$

In the above grammar, $M ::= \overline{D}VD$ means two memory accesses *e1, *e2 are aliased, if the path from *e1 to *e2 consists of a $\overline{D}$-edge from *e1 to e1, a value alias edge V(e1, e2), and a D-edge from e2 to *e2. $V ::= \overline{F}M?F$ means two expressions e1 and e2 are value aliased if there exist two expressions e1' and e2' which are memory aliased, i.e. *M*(e1', e2'), and whose values flow into e1 and e2 respectively, i.e. *F*(e1', e1) and *F*(e2', e2).

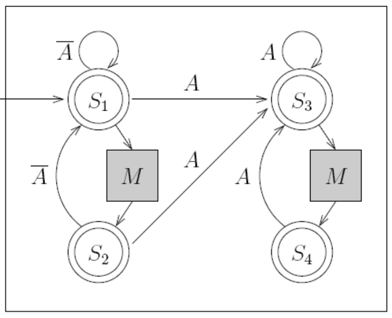
$F ::= (AM?)*$ and $\overline{F} ::= (M?\overline{A})*$ mean that flows of values are due to sequences of assignments and memory aliases. If we eliminate non-terminals, F and $\overline{F}$, we can get:

Memory aliases: $M ::= \overline{D}VD$

Value aliases: $V ::= (M?\overline{A})*M?(AM?)*$



(a) Machine M



(b) Machine V

Figure 1: hierarchical state machine

Thus the memory alias relation and value alias relation can be represented by hierarchical state machines. In figure 1, machine M recognizes memory aliases, machine V recognizes value aliases.

3. Constraint generation and PEG building

Program analysis using constraints is usually divided into two parts: constraint generation and constraint resolution. Constraint generation produces constraints from a program text, gives a declarative specification of the desired information about the program. According to [8], the soundness of analysis can be proven solely on the basis of the constraints generated. So the constraint generation is very critical for both the correctness and precision of set constraints based alias analysis.

As we have mentioned in section 2, in the demand-driven algorithm discussed in this paper, program expression graph (PEG) instead of the constraint graph is used to represent program info, where the both the assignment edges and dereference edges should be added into PEG. Table 1 shows the basic rules of PEG building for intra-procedural analysis for C language.

In table 1, *const* means constant, *var* means variable, *x*, *y*, *f*, *r* represents expressions (including variables, used as operand, result, formal, actual, …), *n*(*e*) represents the node in PEG representing expression *e*. We have two special nodes in PEG, *n_g_ptr*, and *n_g_obj*, representing global pointer and global object separately and *n_g_ptr* always point to *n_g_obj*.

Constraints built from these rules will have much redundancy, since we are not able to identify pointer from the type system of WHIRL. To avoid redundant constraint generation, we apply some tricks to recognize pointer calculation which are already used in current alias analysis in Open64. For example, 1) Results of operations like SQRT, MPY, EQ … can never be value of pointer. 2) If the result type of an operation is pointer type, the result must be value of a pointer. These features can be propagated along WHIRL tree, so as to avoid more redundant constraints to be generated.

CFL formulation of alias analysis can be extended to be field-sensitive by the following grammar, where f_i denotes "field edge". The field edge f_i is an edge in PEG from the node representing the address of a structure to the node representing the address of field i in the structure.

Memory aliases: $M ::= \overline{D}VD$

Value aliases: $V ::= \overline{F}VF \mid \overline{f}_i V f_i \mid M\,?$

Flows of values: $F ::= (AM\,?)*$

$$\overline{F} ::= (M\,?\,\overline{A})*$$

The grammar reflects that two field accesses are aliased if and only if they access the same field (i.e. the structure type and the field id should be the same) and their bases are value aliased. But in reality, the grammar cannot directly apply to ansi-incompliant C programs, where the type casting between pointers of different structure types may make the analysis failed. A feasible way is to apply the similar method presented in [9], i.e. to check whether type-casting operation appears and collapses all the related field access nodes into one.

At present, we assume our applications are ansi-compliant and develop the constraint generation rules for field access, shown the table 2.

Besides F-edge, *A*-edge is also added between structure address and its field address, which is used to reflect the fact that fields always alias with the whole structure.

Grammar	PEG node	PEG *A*-edge	PEG *D*-edge
Const	*n*(*const*) = *n_g_ptr*		
var :: global	*n*(*var*), *n*(&*var*)	*n*(*var*) ← *n_g_ptr* *n*(&*var*) ← *n_g_ptr*	*n*(*var*) ← *n*(&*var*)
var :: formal	*n*(*var*)	*n*(*var*) ← *n_g_ptr*	
var :: Reg/Auto	*n*(*var*)		
&*x*	*n*(*x*), *n*(&*x*)		*n*(*x*) ← *n*(&*x*)
**x*	*n*(*x*), *n*(**x*)		*n*(**x*) ← *n*(*x*)
x = *y*:	*n*(*x*), *n*(*y*)	*n*(*x*) ← *n*(*y*)	
p++/ *p*--	*n*(*p*)		
x = *op*(y_1 …y_n)	*n*(*x*), *n*(y_1), …, *n*(y_n)	*n*(*x*) ← *n*(y_i), i = 1 … n	
x = *allocate*(*y*)	*n*(*x*), *n*(*allocate*), *n*(&(*allocate*))	*n*(*x*) ← *n*(&(*allocate*))	*n*(*allocate*) ← *n*(&(*allocate*))
fun(f_1 … f_n) → (r_1 … r_m) *S**	*n*(f_i), i = 1 … n	*n*(f_i) ← *n_g_ptr*, i = 1 … n	
x_1 … x_m = *p*(y_1 … y_n), assume call-by-value rule here	*n*(x_i), i = 1 … m *n*(y_i), i = 1 … n *n*(*y_i), i = 1 … n	*n*(x_i) ← *n_g_ptr*, i = 1 … m *n*(*y_i) ← *n_g_ptr*, i = 1 … n	*n*(*y_i) ← *n*(y_i) i = 1 … n

Table1: constraint generation and PEG building rules for C grammar

Grammar	PEG node	PEG F-edge	PEG *A*-edge	PEG *D*-edge
$x.f$	$n(x)$, $n(\&x)$, $n(\&x+\mathrm{ofst}(f_i))$, $n(*(\&x+\mathrm{ofst}(f_i)))$,	$n(\&x+\mathrm{ofst}(f_i)) \leftarrow n(\&x)$	$n(\&x+\mathrm{ofst}(f_i)) \leftarrow n(\&x)$	$n(x) \leftarrow n(\&x)$ $n(\&x+\mathrm{ofst}(f_i)) \leftarrow n(*(\&x+\mathrm{ofst}(f_i)))$
$x \rightarrow f_i$	$n(x)$, $n(x+\mathrm{ofst}(f_i))$ $n(*(x+\mathrm{ofst}(f_i)))$	$n(x+\mathrm{ofst}(f_i)) \leftarrow n(x)$	$n(x+\mathrm{ofst}(f_i)) \leftarrow n(x)$	$N(x+\mathrm{ofst}(f_i)) \leftarrow n(*(x+\mathrm{ofst}(f_i)))$

Table2: constraint generation and PEG building rules for struct field

4. Evaluation of demand-driven alias analysis

The demand-driven alias analysis (DDA) implemented in this paper works on WHIRL, and is applied in the same phase as alias classification (AC) in the global optimizer. By keeping the expression id in the POINTS_TO of each memory operation, it can help memory disambiguation in the same way as alias classification.

Compared the AC, the DDA has three advantages: 1) It is based on set inclusion algorithm; 2) The global variable accesses are not collapsed; 3) It is field-sensitive. However, DDA has no linear complexity and it has to give up some of its precision to avoid too much graph traverse time.

Compared to flow free alias analysis (FFA) and flow sensitive analysis (FSA), the advantage of DDA is that it keeps track of the value flow of the whole program (or the whole procedure unit, in case of intra-procedural analysis). The disadvantage of DDA is that it can only supply MAY alias info, but no MUST alias info, while the POINTS_TO (The POINTS_TO structure in Open64) analysis can give definite result, e.g. whether the two memory operations access exactly the same bytes/bits.

In this section, we will evaluate both the precision and the scalability of DDA.

4.1 Precision evaluation

The test cases shown in table 3 include a typical test case to compare the analysis precision between set-union and set-inclusion based alias analysis algorithms and some typical cases from multi-media applications.

The analysis results of DDA, AC, AC+FFA+FSA, and DDA+FFA+FSA are shown in table 4. In the table, we use $x\sim\{y_1, \ldots. y_n\}$ to represent that x alias with $y_1, \ldots, y_n$. The analysis results shown for DDA are got with FFA and FSA turned off. We can see from the results that:

i) For case (1), since DDA is set-inclusion based, it can achieve better precision than AC. Since current DDA is flow-insensitive, it is not as precise as FSA.

ii) For case (2), since DDA can track value flow of the whole procedure, it is able to disambiguate *p from *q and give better result than FSA, which can only track part of the value flow in the program.

iii) For case (3), since DDA is field sensitive, it can disambiguate accesses to different fields when ansi-compliant is assumed. In this case FFA/FSA failed due to loss of high level struct type info in the array.

iv) Since DDA is more precise than AC, we can replace AC with DDA without loss of precision.

4.2 Scalability evaluation

In this section, we evaluate the scalability of DDA using some programs from spec2000. All the experiments are carried out with AC, FFA and FSA turned off. Data in table 5 illustrate the program features and the classification of alias queries generated during Open64 compile process.

In table 5, column 2 ~ 5 show the number of vertices and edges in the program expression graphs, both accumulated count and maximum count are given. Column 6 shows the total number of alias queries from both WOPT phase and CG phase. In our implementation, caching is used to save the alias analysis result after alias analysis for each alias query. Column 7 shows the number of queries which are satisfied by checking cached results. Column 8 shows number of queries which are satisfied by quick analysis, for example, disambiguation of pseudo registers or local variables which are not address taken. Column 9 shows the number of queries which need demand driven analysis.

From table 5, we can see that: 1) The query count is large in Open64; 2) Caching of analysis results is very important. 3) Quick disambiguation is very useful. 4) The part of queries which need deep analysis is small, below 10% for all the cases. But though the number of queries needing deep analysis is small, their results will be cached and used more by later queries. So the precise analysis of these queries is still important.

To avoid unlimited compile time for large programs, a K-limit approach is used, in which a threshold K is used to control the exploration time. The K represents the upper limit of iteration count of the loop in the work-list based algorithm. If the iteration count reaches K, conservative result will be returned.

Table 6 gives the percentage of analysis which is finished in all the demand driven analysis, when K increases from 100 to 10000. We can see that: When K is below 1000, most of the explorations cannot finish. As K increases, more analysis can give definite result. However, most queries cannot get precise result in a reasonable compile time, so the current implementation doesn’t have good scalability.

We believe it may be improved in following ways: 1) Simplify constraint graph as much as possible. 2) Develop more rules which can help do quick analysis. 3) Select simpler underlying algorithm between set-inclusion and set-union, which will be given in the next section.

Test case (1)	Test case (2)	Test case (3)
int foo ()	int a[100]，b[100];	typedef struct {
{	void foo() {	unsigned int bits_left;
int **p, **q;	int i;	unsigned int buffer_size;
int *s1, *s2, *s3;	int *p, *q;	} bitfile;
p = &s1;	p = &a[0];	typedef struct {
p = &s2;	q = &b[10];	char is_leaf;
q = &s3;	for (i=0;i<100;i++) {	char data[4];
q = p;	*p = *q;	} hcb_bin_quad;
p = (int) malloc(100);	p ++;	hcb_bin_quad hcb[10];
q = (int) malloc(100);	q++;	void foo (bitfile *ld, int cb, int n, int b){
return *s1 + *s2 + *s3;	}	for (int i=0; i<n; i++) {
}	}	ld->bits_left += hcb[i].data[b];
		}
		}

Table 3: typical test cases for alias analysis precision evaluation

Test case	Memory operation	DDA	AC	AC + FFA + FSA	DDA+ FFA + FSA

Case (1)	1 S1
	2 s2
	3 s3
	4 *p
	5 *q
	6 *s3
	7 *s1
	8 *s2
		4~{1,2}
		5~{1,2,3,4}
		7~{6}
		8~{6,7}
			4~{1,2,3}
			5~{1,2,3,4}
			7~{6}
			8~{6,7}
				4~{2}
				5~{2}
				7~{6}
				8~{6,7}
					4~{2}
					5~{2}
					7~{6}
					8~{6,7}

Case (2)	1 global_obj
	2 *p
	3 *q
		2~{1}
		3~{1}
			2~{1}
			3~{1,2}
				2~{1, }
				3~{1,2}
					2~{1}
					3~{1}

Case (3)	1 global_obj
	2 ld->bits_left
	3 hcb[i].data[b]
		2~{1}
		3~{1}
			2~{1}
			3~{1,2}
				2~{1}
				3~{1,2}
					2~{1}
					3~{1}

Table4: alias analysis results comparison

Test case	Summary of PEGs				Query count	By cache	By analysis count	
	node count		A-edge count				quick analysis	DDA analysis
	Total	Max	total	max				
Swim	391	119	568	208	23081	75%	18%	7%
Mgrid	712	132	1019	247	62410	61%	36%	3%
Equake	1899	280	1761	362	98668	45%	52%	3%
Art	1749	95	1643	122	17573	27%	65%	8%

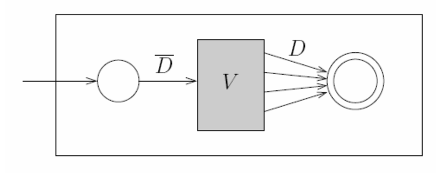
Table5: program features and alias queries classification

Test case	K = 100	K = 1000	K = 5000	K = 10000
Swim	21%	21%	36%	100%
Mgrid	29%	32%	76%	100%
Equake	23%	31%	38%	45%
Art	11%	42%	79%	100%

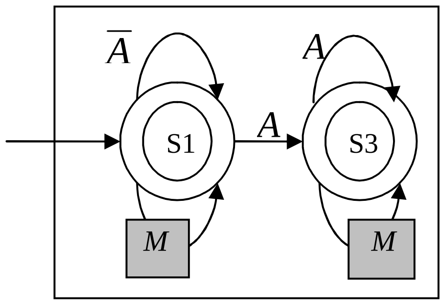
Table6: percentage of queries finished for given K-limit

5. One-level flow demand driven analysis

In the original state machine, the machine M is not transitive. To get better scalability, we now make M transitive, so the four states in machine V can be changed into two, shown in figure 2.



(a) Machine M



(b) Machine V

Figure 2: one-level flow hierarchical state machine

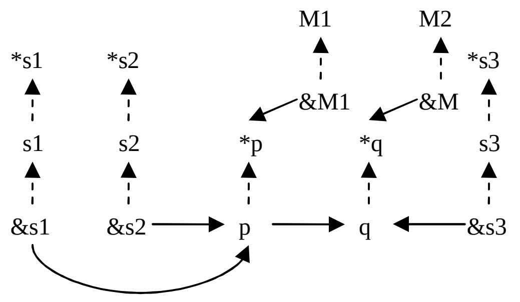
Based on this state machine, we can developed a new demand driven alias analysis and achieve the same precision as Das's one-level flow algorithm. It can be understood in this way: if *e1 can reach *e2 by M(*e1, *e2), and *e2 can reach *e3 by M(*e2, *e3), then *e1 will also reach *e3 by continuously following M(*e1, *e2) and M(*e2, *e3). That means the expressions connected by memory alias relationship will all carry the same value, so as to point to the same object.

Thus we can union these expressions into one "value alias group" (i.e. these expressions have the same value/content, though may access different locations), then merge dereferences of these expressions into one PEG node (i.e. same value/content and same location). We use a special node as representative of the value alias group, which we call "value holder", and then the value flows between the nodes inside the "value alias group" need not to be checked in future analysis, while the value flows outside the group will only be carried on the "value holder". That means the assignment edges inside the group will be deleted while the edges between nodes inside the group and those outside the group will be moved to lie between the "value holder" and those outside nodes. Since both the node count and edge count in PEG are reduced, later analysis will be greatly simplified.

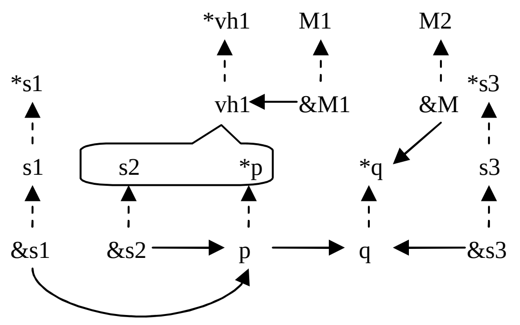
Figure 3 gives an example of online PEG simplification for the case (1) given in table 3, where solid lines represent A-edges and dash lines represent D-edges. The nodes surrounded by the callout form a value alias group. We can see that in the new algorithm, value alias relationship is still exploited by reachability analysis, but when memory alias is found, we will perform PEG (program expression graph) simplification of the related nodes. So the algorithm is demand driven in two folds: 1) demand driven reachability analysis, 2) demand driven PEG simplification.

Compared with SSA, the "value holder" here corresponds to the phi node in SSA. The phi node factors the value flows in and out of control flow structure, while "value holder" factors the value flows in and out of "value alias group".

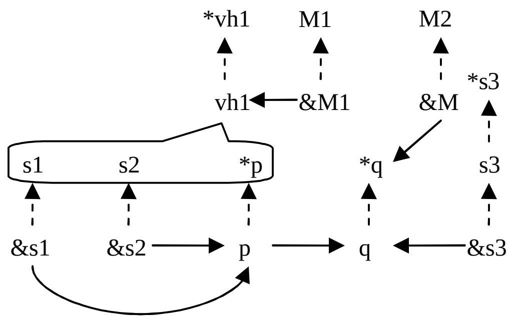
Figure 4 shows the detailed algorithm description, where *val_alias_g*(*n*) means to find the value alias group for node *n*, *val_holder*(*g*) means to find the value holder node of the value alias group *g*, *val_holder*(*n*) means to find the value holder node of the value alias group that include node *n*, and *peg_node*(*e*) means to find the corresponding node in PEG for expression *e*.



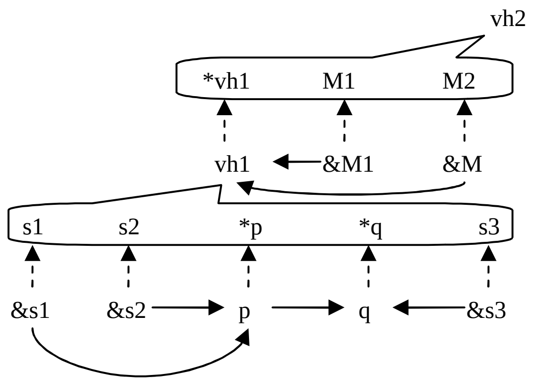
(a) Initial PEG



(b) PEG after memory alias of *p and s2 was found



(c) PEG after memory alias of *p and s1 was found



(d) Final PEG

Figure 3: online PEG simplification

```
MAYALIAS (e1 : Expr, e2 : Expr)
/* initialize worklist */
w ← { <addr(e1), addr(e1), S1> }
while (w is not empty)
 remove <n, s, c> from w
 /* check if the destination has been reached */
 if (peg_node(addr(e1)) == peg_node(addr(e2)))
  then return true
 else if (((n == addr(e1)) ^ (s == addr(e2))) ||
        ((n == addr(e2)) ^ (s == addr(e1))))
  then return true
 /* propagate information upward */
 ds ← deref(s)
 dn ← deref(n)
 if ( (dn != null) ^ (ds!= null) ^ (dn != ds) ^
    (val_alias_g (dn) != val_alias_g(ds)))
  then g ← UNION (val_alias_g(dn), val_alias_g(ds))
     merge all the derefs of nodes in g
 /* propagate reachability through value flows */
 vn ← val_holder(n)
 vs ← val_holder(s)
 switch (c)
 case S1 :
  for each m in assign_from(vn) :
    PROPAGATE(w, m, vs, S1)
  for each m in assign_to(fn) :
    PROPAGATE(w, m, vs, S3)
 case S3 :
  for each m in assign_to(vn) :
    PROPAGATE(w, m, vs, S3)
 /* propagate information downward */
 if (n == val_holder(n))
  then for each m in val_alias_g(n)
   if (addr(m) != null)
     then PROPAGATE(w, addr(m), addr(m), S1)
   else if (addr(n) != null)
     then PROPAGATE(w, addr(n), addr(n), S1)


 return false
```

Figure 4: one-level flow demand driven algorithm

```
PROPAGATE (w, n, s, c)
 if ( <s, c>  ∉  reach(n))
  then reach(n) ← reach(n) ∪ { <s, c> }
       w ← w ∪ {<n, s, c>}
UNION (g1, g2)
 g0 ← union(g1, g2)
 for each m in g0
   for each e in out_edges(m)
     delete e
     if (target(e)  ∉  g0)
       then generate edge(val_holder(g0), target(e))
   for each e in in_edges(m)
     delete e
     if (source(e)  ∉  g0)
       then generate edge(source(e), val_holder(g0))
```

Figure 4: one-level flow demand driven algorithm (continue)

6. Evaluation of one-level flow demand-driven alias analysis

In this section, we compare the scalability and precision of the original demand driven alias analysis given in [7] with our one-level flow demand driven alias analysis.

Data shown in table 7 give the percentage of the finished analysis in all the analysis, when K increases from 100 to 5000. The columns marked by "DDA" show the data for original demand driven algorithm, while those marked by "Olf DDA" show the data for our one-level flow demand driven algorithm.

We can see that even when K is very small (e.g. 100), one-level flow DDA can already finish about half of all the analysis for all the cases, which DDA cannot achieve when K is extended to 5000 for some cases.

Although DDA in theory should have the same precision as Andersen's algorithm, it cannot give such precise results in reality, where reasonable compile time is needed. Table 8 shows the data about the percentage of analysis which give "not aliased" results. It is easy to understand that the higher percentage, the more precise. We can see that for most cases, one-level flow DDA can give much more precise results than just DDA alone.

Test case	K = 100		K = 1000		K = 5000	
	DDA	Olf DDA	DDA	Olf DDA	DDA	Olf DDA
Swim	21%	54%	21%	67%	36%	100%
Mgrid	29%	58%	32%	92%	76%	100%
Equake	23%	44%	31%	58%	38%	66%
Art	11%	41%	42%	77%	79%	100%

Table 7: percentage of analysis finished

Test case	K = 100		K = 1000		K = 5000	
	DDA	Olf DDA	DDA	Olf DDA	DDA	Olf DDA
Swim	21%	53%	21%	66%	35%	99%
Mgrid	27%	54%	31%	88%	77%	97%
Equake	19%	22%	20%	27%	20%	30%
Art	3%	10%	13%	31%	35%	54%

Table8: percentage of not-aliased results

7. Conclusions and Future work

We implemented the demand-driven alias analysis presented in [7]. After evaluating both precision and scalability, we found that it is able to give good precision for C programs, but still has scalability issue when used in a product compiler.

To improve the scalability of the algorithm, we developed a one-level flow demand driven algorithm. We showed that the new algorithm gives much more precise results for a reasonable compile time and have better scalability.

Our future work will be to extend the current implementation and try to use it in inter-procedural analysis.

Acknowledgements

The authors would like to thank Sun Chan for his encouragement, support and guidance. We also thank Shin Ming Liu and Shuxin Yang for their valuable advices on many complex problems in alias analysis work.

References

[1] Steenguaard, Points-to analysis in almost linear time, Principles of Programming Languages, 1995.

[2] Lars Ole Andersen, Program analysis and specialization for the C Programming Language, PhD thesis, 1994.

[3] Manuvir Das, Unification-based pointer analysis with directional assignments, Programming Language Design and Implementation, 2000.

[4] Manuel Fahndrich, Jeffrey S. Foster, Zhendong Su, Alexander Aiken, Partial online cycle elimination in inclusion constraint graphs, Programming Language Design and Implementation, 1998.

[5] Atanas Rountev, Satish Chandra, Off-line variable substituion for scaling points-to analysis, Programming Language Design and Implementation, 2000.

[6] Heintz and Tardieu, Demand driven pointer analysis, Programming Language Design and Implementation, 2001.

[7] Xin Zheng and Rugina, Demand driven Alias analysis for C, Principles of Programming Languages, 2008.

[8] Aiken, Introduction to set constraint-Based program analysis, Science of Computer Programming 1999.

[9] Chris, LLVM: An Infrastructure for multistage optimization, Master thesis 2002.