Interprocedural Modification Side Effect Analysis With Pointer Aliasing*

William Landi

Siemens Corporate Research Inc
755 College Rd. East
Princeton, NJ 08540
landi@cadillac.siemens.com

Barbara G. Ryder
Sean Zhang

Department of Computer Science
Rutgers University
Hill Center, Busch Campus
New Brunswick, NJ 08903
ryder@cs.rutgers.edu
xxzhang@cs.rutgers.edu

Abstract

We present a new interprocedural modification side effects algorithm for C programs, that can discern side effects through general-purpose pointer usage. Ours is the first complete design and implementation of such an algorithm. Preliminary performance findings support the practicality of the technique, which is based on our previous approximation algorithm for pointer aliases [LR92]. Each indirect store through a pointer variable is found, on average, to correspond to a store into 1.1 locations. This indicates that our program-point-specific pointer aliasing information is quite precise when used to determine the effects of these stores.

1 Introduction

Accurate compile-time calculation of possible interprocedural side effects is crucial for aggressive compiler optimization [ASU86], practical dependence analysis in programs with procedure calls [Ban88, BC86, Wol89], data-flow based testing [RW82, OW91], incremental semantic change analysis of software [Ryd89], interprocedural def-use relations [PRL91, PLR92] and effective static interprocedural program slicing [HRB88, OO84, Ven91, Wei84]. These are key problems in parallel and sequential programming environments; the utility of tools to solve these problems is directly dependent on the accuracy of the data flow information available to them. We need a efficient method to report program-point-specific data flow information for these applications. Existing techniques for Fortran can not supply this information; they only handle call-by-reference induced aliasing and are insufficient for languages with general-purpose pointer usage.

Interprocedural modification side effects were first handled by Allen for acyclic call multigraphs [All74, Spi71]. Later, Barth explored the use of relations to capture side effects in recursive programs [Bar78]. Banning [Ban79] first noted the decomposition of the problem for Fortran (and other

*The research reported here was supported, in part, by Siemens Corporate Research and NSF grants CISE-CCR-92-08632 and CCR-9023628 1/5.

languages where aliasing is imposed only by call-by-reference parameter passing); he separated out two flow insensitive calculations on the call multigraph: one for side effects and a separate one for aliases. Cooper and Kennedy [Coo85, CK84, CK87] further decomposed the problem into side effects on global variables and side effects accomplished through parameter passing. Burke showed that these two subproblems on globals and formals can be solved by a similar problem decomposition [Bur90].

Choi, Burke, and Carini mention a modification side effects algorithm for languages with pointers based on their flow insensitive pointer aliasing calculation [CBC93]; it is difficult to compare our work to theirs, because they give no description of their algorithm. Clearly, the importance of work in this area must be judged by algorithm performance in practice, in terms of precision and resource needs.

In this paper, we present the first design and implementation of an interprocedural modification side effects algorithm for languages with general-purpose pointers (e.g., C); this is the first such algorithm to use program-point-specific aliasing information. Our algorithm reports program-point-specific possible modification side effects (i.e., MOD); our results are more precise than information derivable using the same alias summary for all statements of a procedure. Our algorithm is based on an initial interprocedural pass that computes a flow sensitive approximation of program-point-specific pointer-induced aliases [LR92]. These are used to gather procedure summary modification information, with subsequent flow insensitive propagation of modifications through the program call multigraph. Finally, call site modification information is calculated using the results of the procedure side effects summary.

We have implemented our MOD algorithm as a back-end analysis on our pointer aliasing implementation [LR92]. Initial experiments have been run with ten of the programs which appeared in [LR92]. The side effects reported are differentiated by location type: global, local, dynamically-created, and not-visible (within that procedure).[1] Measurements of average and maximum number of side effects found per procedure, per call site, and per assignment statement (i.e., $*p =$) have been made, as well as calculations of analysis times and the relative extra cost imposed by using our conditional analysis technique [LR92].

Most importantly, our results over the ten programs show that on average 1.1 locations are assigned values per assignment statement, indicating that often there is only one alias for a deref-

[1] *Not-visibles* are local variables of other procedures or an earlier instantiation of the current procedure [LR92].

erenced pointer variable at a program point (i.e., $*p =$). Also, on average, less than 9% of all the visible variables at each assignment statement are assigned values; this result indicates that our pointer aliasing is very precise, because we are not overestimating the effects of the assignments by reporting many spurious aliases.

In this abstract, Sections 2 and 3 discuss our pointer aliasing algorithm and present our decomposition of the modification side effects problem, contrasting it to the Fortran analysis. Section 4 reports our implementation results in detail. Section 5 summarizes the contributions of this work. Appendix A gives an example of our analysis.

2 Pointer Aliasing Analysis

Iterative data flow analysis is a fixed point calculation for recursive equations defined on a graph representing a program, that safely approximates the *meet over all paths solution* [Hec77] for the graph. For interprocedural data flow, not all paths in the obvious graph representation correspond to real program executions. A *realizable* path is a path on which every procedure returns to the call site which invoked it[LR92]. Paths on which a procedure does not return to the call site which invoked it, are unrealizable and can never happen in an actual execution. A fundamental problem of interprocedural analysis is how to restrict the propagation of data flow information to realizable paths.

Jones and Muchnick[JM82] give a general approach for handling this problem for *abstract interpretations*, which is also valid for data flow analysis. They associate with each data flow fact, an abstraction of the run-time stack on paths on which the fact is created. This abstraction, created at procedure entry, is used at procedure exit to determine to which call site(s), the data flow information should be propagated. Our *conditional aliasing* approach [LR91, LR92] can be seen as an application of this idea. The data flow fact that x and y are aliased at program point n is represented by an unordered pair $\langle$x,y$\rangle$ at n. Our encoding of the run-time stack is the set of *reaching aliases*[2] (RA) that exists at entry of procedure p containing n when p is invoked. RA can be used to determine to which call sites, aliases should be propagated. Details can be found in [LR92], but at a high level:

[2]Reaching aliases were referred to in [LR92] by the term *assumed aliases*.

If the alias solution at a call site (C) is sufficient to create RA at entry of the called procedure,[3] *then any alias associated with RA at the exit of the called procedure is valid at C.*

In [LR92], we safely restrict the size of the alias sets (RA) to one, yielding a compact and effective encoding of the run-time stack.[4] Choi et. al. [CBC93] also use alias sets, called *source alias sets*, as part of their encoding of the run-time stack in their flow sensitive aliasing algorithm, but do not restrict the set size. It will be interesting to see what effects this has on precision and performance of their algorithm in practice.

Our MOD algorithm can be described independent of the details of the abstraction for the run-time stack. In this paper, we use our abstraction, reaching alias sets of size one. $Calias(n, RA)$ represents the set of *may aliases* at program point n for reaching alias RA, obtained by our approximation alias algorithm [LR92][5].

3 Decomposition of the MOD problem

We are solving for MOD(s), the set of variables possibly modified by execution of the statement at program point s. To elaborate on our approach, first we solve the conditional aliasing problem. Given the results of this alias analysis, we calculate the two related problems (i.) *PMOD*, a procedure-level summary of conditional modification side effects which can occur, given a specific reaching alias condition at procedure entry, and (ii.) *CMOD*, a set of conditionally modified locations at each program point corresponding to a specific reaching alias. *CMOD* can then be used to derive MOD.[6] We assume call-by-value parameter passing as in C; call-by-reference parameter passing can be accommodated as well. (An explanation will appear in the full paper).

In our analyses, we report modifications to *fixed-locations*; these are either user-defined variable names or heap storage creation site names. For example in C syntax, x and $x.f$ are fixed-locations whereas $*p$ and $p \rightarrow f$ are not. We have named each dynamic allocation site, similar to [RM88]. Each dynamically allocated fixed-location is identified by the site that created it; therefore, we cannot distinguish between two fixed-locations created at the same site.

[3] Ignoring parameter bindings, this is simply $RA \subseteq$ alias solution at call site.

[4] Use of this encoding yields a precise solution for aliasing in the presence of one level of dereferencing; for multiple levels of dereferencing, this yields a safe approximate solution for aliasing [LR91, LR92].

[5] In [LR92], we used *may-holds* to represent conditional aliasing information. $Calias(n, RA) = \{PA \mid may\text{-}holds(n, RA, PA)\}$.

[6] *PMOD* can similarly be used to derive a procedure-level summary of modification side effects over all calls to that procedure.

We are using a decomposition approach, dividing the MOD problem into subproblems that are individually easier to solve than the monolithic problem. Thus, we are continuing in the tradition established in the analysis of Fortran. To present our decomposition, we need the following notation:

- *DIRMOD*(n) is the left hand side of the assignment at program point n in procedure *proc*[7].
- *Predecessors*(n) is the set of predecessors of n in the program.

Following the same approach as aliasing, our modification side effects sets are also associated with our encoding of the run-time stack, reaching aliases, to restrict our attention to realizable paths. *CondLMOD*(n,*RA*) is the set of fixed-locations modified by the assignment at n because of aliases that occur at any of the predecessors of n[8] when RA reaches the entry of *proc*. Fixed-locations that are directly modified by the assignment are modified regardless of reaching aliases. We use the reaching alias ϕ to represent this condition. Thus, *DIRMOD*(n) is added to *CondLMOD*(n, ϕ).

$$CondLMOD(n, RA) = \{ \, obj \, | obj \in \mathcal{S} \cup \mathcal{R} \ and \ obj \text{ is a fixed-location}\}$$

$$\text{where} \quad \mathcal{S} = \bigcup_{pred \ \in \ Predecessors(n)} \left\{ obj_1 \, \middle| \, \begin{array}{c} obj_2 \in DIRMOD(n) \ \ and \\ \langle obj_1, obj_2 \rangle \in Calias(pred, RA) \end{array} \right\}$$

$$\mathcal{R} = \left(\begin{array}{cl} DIRMOD(n) & \text{if } RA = \phi \\ \emptyset & \text{if } RA \neq \phi \end{array} \right)$$

For a procedure P and reaching aliases RA, *CondIMOD*(P, RA) contains the fixed-locations modified by assignments in procedure P.

$$CondIMOD(P, RA) = \bigcup_{n \text{ an assignment in } P} CondLMOD(n, RA)$$

PMOD(P, RA) is the set of fixed-locations modified by procedure P, including the effects of calls from within P, considering only aliases conditional on reaching alias RA. The *PMOD* sets for a procedure summarize its modification side effects for a given reaching alias. They are specified by the following, possibly recursive, system of data flow equations which can be solved iteratively.

[7] In our decomposition, *assignment* is synonymous with value-setting statement. Thus, DIRMOD(n) captures the direct effects of all value-setting statements.

[8] We are assuming *on bottom* alias information, i.e., information at a statement incorporates the alias effects of that statement.

In the *PMOD* equation, $call_Q$ is a call site in P at which P calls Q. The function b_{call_Q}, specific to the call $call_Q$, maps names from the called procedure (Q) to the calling procedure (P) according to scoping rules [CK87][9]. $call(call_Q, RA)$ represents the set of reaching aliases at the entry of Q induced by the parameter bindings and aliases at $call_Q$ given RA reached the entry of P. (b_{call_Q} and *call* will be discussed in more detail in the full paper.)

$$PMOD(P, RA) = CondIMOD(P, RA) \cup \bigcup_{call_Q \; in \; P} \{ \; obj \mid obj \in \mathcal{S} \; and \; obj \text{ is a fixed-location } \}$$

$$\text{where} \quad \mathcal{S} = \bigcup_{RA' \in \; call(call_Q, RA)} b_{call_Q}(PMOD(Q, RA'))]$$

We now specify the modification side effects for calls and assignments.

$$CMOD(n, RA) = \begin{cases} CondLMOD(n, RA) & \text{if } n \text{ is an assignment} \\ \{ \; obj \mid obj \in \mathcal{S} \; and \; obj \text{ is a fixed-location } \} & \text{if } n \text{ is a call of Q} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\text{where} \quad \mathcal{S} = \bigcup_{RA' \; \in \; call(n, RA)} b_n(PMOD(Q, RA'))$$

Finally, $MOD(n)$ summarizes the side effects over all executions of n in procedure P and $MOD(P)$ summarizes the side effects over all calls of P. Both are obtained by considering all reaching aliases for P.

$$MOD(n) = \bigcup_{reaching \; alias \; RA \; for \; P} CMOD(n, RA)$$

$$MOD(P) = \bigcup_{reaching \; alias \; RA \; for \; P} PMOD(P, RA)$$

The precision of our MOD solution depends largely on the precision of the underlying alias analysis. Furthermore, when the alias algorithm is safe (i.e., erring conservatively [LR92]), our MOD decomposition is also safe. Our decomposition of the MOD problem is pictured in figure 1. (Further comparison of our decomposition to those of Cooper and Kennedy and Burke will be included in the final paper.) In Appendix A, we show *PMOD* and *CMOD* solutions for an example program.

[9] In particular, b_{call_Q} handles the mapping of *not-visible* fixed-locations.

The worst-case time complexity of our MOD algorithm is $O(N_{RA}^2 * V^2 * N_c + A + N_a * C_u)$. N_c and N_a are the number of calls and assignments in the program, respectively. N_{RA} is the maximum number of reaching aliases at the entry of any procedure. V is the number of fixed-locations. A is the total number of conditional aliases in the program and C_u is the cost of the union operation over sets of fixed-locations. (This will be elaborated further in the full paper.)

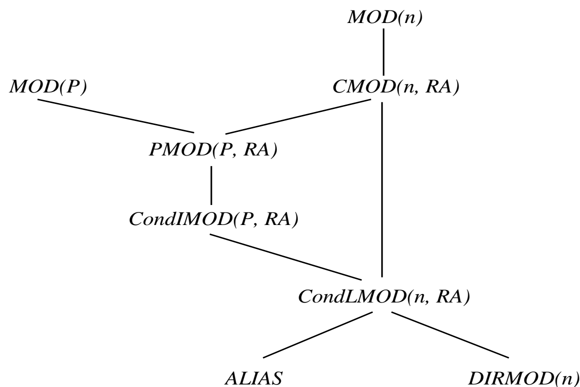

Figure 1: Decomposition of the MOD problem

4 Empirical Results

We have implemented our MOD decomposition and have empirical results for ten of the programs analyzed in [LR92]. Our implementation is written in C and analyzes a reduced version of C that excludes: union types, casting[10], pointers to functions, and exception handling. The first two of these omissions are not theoretically difficult to handle, but complicate the implementation and must be addressed before we can study a broader base of programs. We allow arrays and pointer arithmetic; however, we simply treat arrays as aggregates.

The ten programs we have analyzed and their sizes are in figure 2. In this, and all subsequent figures, the programs are sorted by size in lines of code. In figure 2, the reported MOD times *do not include* the alias times, so the total analysis time is the sum of the two columns. In general, the MOD times are an order of magnitude smaller then the alias times. Because of several implementation optimizations, the alias times reported here are better than those we reported in [LR92], even with our addition of more sophisticated tracking of dynamic storage. For the larger programs, our current implementation is about 2-3 times faster.

[10]The only casting we handle is simple casting for p = *malloc()*.

program	lines of code	number of procedures	number of calls	number of assignments	MOD time (sec)	alias time (sec)
allroots	188	8	20	64	0.04	0.34
diffh	268	16	51	100	0.14	1.00
fixoutput	458	8	14	47	0.07	0.60
ul	541	20	75	239	0.96	4.21
lex315	776	19	104	130	0.39	1.58
loader	1539	33	86	248	1.23	15.86
diff	1782	46	166	621	2.22	14.54
football	2354	61	265	820	1.09	3.36
assembler	3361	55	256	503	4.27	112.23
simulator	4663	109	433	720	1.98	21.92

Figure 2: Program size and analysis times

In figure 3, we give summary statistics for the MOD solution for assignment statements. These statistics are subdivided with respect to the type of fixed-locations being modified. There are five types:

- **glo**: MOD information for global variables.
- **dyn**: MOD information for dynamic storage locations.
- **loc**: MOD information for local variables of the enclosing procedure.
- **nv**: (not-visible) MOD information for local variables of *other* procedures or of an earlier recursive instantiation of the enclosing procedure.
- **tot**: MOD information for all fixed-locations.

We give three different summary statistics. **Average/assignment** (**maximum/assignment**) is the average (maximum) number of fixed-locations modified by assignment statements. **Average percent/assignment** is more complicated. We define the number of fixed-locations potentially modified by an assignment as the sum of:

- number of globals in the program
- number of dynamic allocation sites
- number of locals in the enclosing procedure
- number of locals of other procedures[11] accessible through globals and formals at the entry of the enclosing procedure

[11] plus locals of earlier recursive instantiations of this procedure

The **percent/assignment** is simply the number of fixed-locations modified, divided by the number potentially modified locations of the appropriate type per assignment. The **average percent/assignment** is the average of **percent/assignment** over all assignments. For some assignments the number of possible locals and the number of not-visibles are zero. In these cases, we use "0%" as the **percent/assignment**.

The results in figure 3 are extremely encouraging. Any executable assignment in a normally terminating program will modify at least one fixed-location. Thus, 1.0 is a lower bound of **tot** for **average/assign**. The **tot** column of **average/assign** contains mostly 1.0s and a maximum value of 1.3. This indicates that our algorithm is highly precise. (In the full paper, we will present empirical evidence on the precision of our calculation in the manner we did for our aliasing algorithm[LR92].) **Average percent/assign** indicates how much more precision is obtained from our MOD calculation in comparison to using the worst-case assumption that all fixed-locations are modified. The **tot** column runs from 1% to 9% indicating that it is worth doing our MOD calculation. **Maximum/assign** is interesting, but can not easily be used to justify the quality of our calculation.

Figure 4 has the same structure as figure 3 and is also encouraging; however, it is harder to get a good lower bound on how many fixed-locations are modified for procedures. We think the numbers reported are surprisingly small. It seems likely that a procedure would modify all of its locals and thus you would expect **average percent/procedure** for **loc** to be 100%. We do not see this value because some procedures do not have any locals; these procedures introduce a 0% into the average calculation. (In the full version of the paper, we will present similar results for call sites.)

Finally, we present empirical evidence that by associating reaching aliases (RA) with *CMOD* and *PMOD*, we are not incurring much unnecessary work in our algorithm. This would be possible, if some assignment (or call or procedure) modifies the same fixed-location under many different reaching aliases, since, to solve the MOD problem, we are only interested in which fixed-locations are possibly modified, not in which conditions lead to their possible modification. To verify that our approach does not involve duplicate effort, we compare the size of the MOD solution to the size of the *CMOD* and *PMOD* solutions for each procedure in figure 5. If there were redundant calculations, then the ratio of the *CMOD* (and *PMOD*) size to the MOD size would attest to this phenomenon. As seen in columns (3) and (6), the ratios are all close to one which indicates that

program	Average/assign glo	dyn	loc	nv	tot	Average percent/assign glo	dyn	loc	nv	tot	Maximum/assign glo	dyn	loc	nv	tot
allroots	0.1	0.0	0.9	0.0	1.0	2%	3%	22%	0%	9%	1	1	1	0	1
diffh	0.2	0.0	0.7	0.0	1.0	2%	1%	20%	0%	6%	1	1	1	0	1
fixoutput	0.7	0.2	0.2	0.0	1.1	6%	6%	6%	0%	6%	1	2	1	0	2
ul	0.5	0.0	0.5	0.0	1.0	1%	0%	11%	0%	3%	1	0	1	0	1
lex315	0.2	0.1	0.8	0.0	1.0	2%	3%	10%	0%	5%	1	2	1	0	2
loader	0.1	0.1	0.8	0.2	1.2	<1%	<1%	17%	4%	3%	1	2	1	9	9
diff	0.2	0.0	0.7	0.0	1.0	<1%	<1%	15%	0%	1%	2	1	1	0	2
football	0.2	0.0	0.7	0.0	1.0	<1%	0%	16%	<1%	1%	3	0	1	1	3
assembler	0.2	0.0	0.7	0.4	1.3	<1%	<1%	19%	6%	3%	2	1	1	9	10
simulator	0.2	0.1	0.6	0.2	1.2	<1%	<1%	25%	6%	2%	1	2	1	13	13

Figure 3: MOD statistics for assignment statements

program	Average/procedure glo	dyn	loc	nv	tot	Average percent/procedure glo	dyn	loc	nv	tot	Maximum/procedure glo	dyn	loc	nv	tot
allroots	1.5	0.6	2.8	0.0	4.9	25%	63%	75%	0%	50%	6	1	6	0	7
diffh	2.4	0.4	1.8	0.0	4.6	18%	44%	50%	0%	28%	13	1	7	0	14
fixoutput	5.1	2.4	1.0	0.0	8.5	46%	79%	25%	0%	56%	11	3	4	0	14
ul	6.5	0.0	1.6	0.0	8.2	19%	0%	50%	0%	22%	35	0	6	0	35
lex315	3.1	1.7	1.6	0.0	6.4	25%	58%	26%	0%	38%	12	3	16	0	16
loader	1.7	2.4	2.5	1.0	7.5	12%	22%	57%	45%	23%	14	11	18	9	36
diff	5.2	1.1	3.0	0.0	9.3	8%	12%	74%	0%	12%	63	9	11	0	72
football	6.0	0.0	2.9	0.0	8.9	9%	0%	69%	10%	12%	69	0	36	1	69
assembler	4.7	1.7	2.7	1.7	10.9	15%	28%	67%	32%	23%	31	6	13	19	37
simulator	2.6	0.8	1.9	0.8	6.0	8%	7%	79%	81%	13%	33	11	7	24	44

Figure 4: MOD statistics for procedures

program	(1) MOD size at calls and assignments	(2) *CMOD* size at calls and assignments	(3) $\frac{(2)}{(1)}$	(4) MOD size of procedures	(5) *PMOD* size of procedures	(6) $\frac{(5)}{(4)}$
allroots	79	79	1.0	39	39	1.0
diffh	173	173	1.0	74	74	1.0
fixoutput	147	150	1.0	96	98	1.0
ul	353	353	1.0	163	163	1.0
lex315	571	578	1.0	122	127	1.0
loader	490	490	1.0	249	251	1.0
diff	813	813	1.0	426	426	1.0
football	1595	1595	1.0	792	792	1.0
assembler	1429	1570	1.1	597	718	1.2
simulator	1534	1534	1.0	655	657	1.0

Figure 5: Solution sizes

little redundant work is being performed.

5 Conclusions

We have presented the design and implementation of a new interprocedural side effects algorithm for languages that allow general-purpose pointer usage (e.g., C). Our algorithm is based on our conditional analysis approach, that already has been used successfully in the approximation of pointer-induced aliases [LR92] and interprocedural reaching definitions [PRL91, PLR92]. Preliminary results from our prototype implementation indicate that our algorithm is practical, efficient and quite accurate. Future work includes broadening the class of C programs handled by our prototype, making our algorithms incremental, and scaling up to handle large C systems.

References

[All74] F. E. Allen. Interprocedural data flow analysis. In *Proceedings of 1974 IFIP Congress*, pages 398–402, Amsterdam, Holland, 1974. Institute of Electrical and Electronics Engineers, Inc., North Holland Publishing Company.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Ban79] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 29–41, January 1979.

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, MA, 1988.

[Bar78] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[BC86] M. Burke and R. Cytron. Interprocedural depenodence analysis and parallelization. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 162–175, June 1986. SIGPLAN Notices, Vol 21, No 6.

[Bur90] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[CBC93] Jong-Doek Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. January 1993. to appear in Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages.

[CK84] K. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 247–258, June 1984. SIGPLAN Notices, Vol 19, No 6.

[CK87] K. Cooper and K. Kennedy. Complexity of interprocedural side-effect analysis. Computer Science Department Technical Report TR87-61, Rice University, October 1987.

[Coo85] K. Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 281–290, January 1985.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

[HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 35–46, June 1988.

[JM82] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 66–74, January 1982.

[LR91] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, January 1991.

[LR92] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[OO84] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, May 1984.

[OW91] T. J. Ostrand and E. Weyuker. Data flow based test adequecy analysis for languages with pointers. In *Proceedings of the 1991 Symposium on Software Testing, Analysis and Verification (TAV4)*, October 1991. Victoria, B.C., Canada.

[PLR92] H. D. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations in the presence of single level pointers. Laboratory for Computer Science Research Technical Report LCSR-TR-193, Department of Computer Science, Rutgers University, 1992. submitted for publication.

[PRL91] H. Pande, B. G. Ryder, and W. Landi. Interprocedural def-use associations for C programs. In *Proceedings of the ACM SIGSOFT Conference on Testing, Analysis and Validation*, pages 139–153, October 1991.

[RM88] C. Ruggieri and T. Murtagh. Lifetime analysis of dynamically allocated objects. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 285–293, January 1988.

[RW82] S. Rapps and E. Weyuker. Data flow analysis techniques for program test data selection. In *Proceedings of the Sixth International Conference on Software Engineering*, pages 272–278, September 1982.

[Ryd89] B. G. Ryder. Ismm: Incremental software maintenance manager. In *Proceedings of the IEEE Computer Society Conference on Software Maintenance*, pages 142–164, October 1989.

[Spi71] T. Spillman. Exposing side effects in a PL-I optimizing compiler. In *Proceedings of IFIPS Conference*, pages TA–3–56:TA–3–62, 1971.

[Ven91] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 107–119, June 1991.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

A An Example

We show the results of our analysis for the example program in figure 6. The program is represented in an intermediate form called ICFG [LR92]. Both *main* and R are analyzed with reaching alias ϕ at their entries. The first call to R makes the alias <*b,x> reach the entry of R. The second call to R creates the alias <*b,y> at the entry. R is analyzed for each of these aliases. There are no aliases in *main* and the alias solution for R is shown in figure 7. The *PMOD* and *CMOD* solutions computed according to our decomposition are in figure 8 and figure 9 respectively. Empty entries in these tables mean either no alias or no side effect.

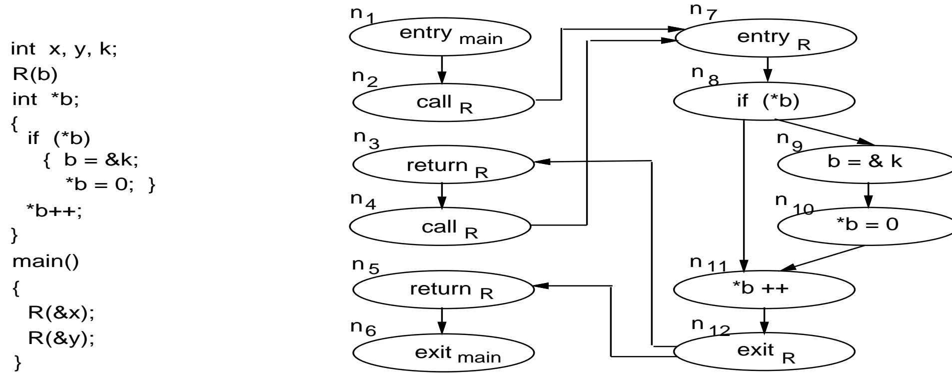

Figure 6: An example program and its ICFG

Reaching Alias	Alias Solutions for Procedure R					
	n_7	n_8	n_9	n_{10}	n_{11}	n_{12}
ϕ			<∗b,k>	<∗b,k>	<∗b,k>	<∗b,k>
<∗b,x>	<∗b,x>	<∗b,x>			<∗b,x>	<∗b,x>
<∗b,y>	<∗b,y>	<∗b,y>			<∗b,y>	<∗b,y>

Figure 7: Alias solutions for the example program

Reaching Alias	*PMOD* Solutions for *main*
ϕ	{ x, k, y }

Reaching Alias	*PMOD* Solutions for R
ϕ	{ k, b }
<∗b,x>	{ x }
<∗b,y>	{ y }

Figure 8: *PMOD* solutions

Reaching Alias	*CMOD* Solutions for *main*					
	n_1	n_2	n_3	n_4	n_5	n_6
ϕ		{ x, k }		{ y, k }		

Reaching Alias	*CMOD* Solutions for R					
	n_7	n_8	n_9	n_{10}	n_{11}	n_{12}
ϕ			{ b }	{ k }	{ k }	
<∗b,x>					{ x }	
<∗b,y>					{ y }	

Figure 9: *CMOD* solutions