Interprocedural Modification Side Effect Analysis With Pointer Aliasing*

William Landi

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

Barbara G. Ryder
Sean Zhang

Department of Computer Science
Rutgers University, 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 1.2 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 an efficient method to report program-point-specific data flow information for these applications. Existing techniques for FORTRAN cannot 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 languages where aliasing is imposed only by call-by-reference parameter passing); he separated out two flow insensitive[1] 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

*The research reported here was supported, in part, by Siemens Corporate Research and NSF grants CISE-CCR-92-08632 and CCR-9023628 2/5.

ACM-SIGPLAN-PLDI-6/93/Albuquerque, N.M.
© 1993 ACM 0-89791-598-4/93/0006/0056...$1.50

[1] We say an interprocedural data flow problem is *flow sensitive* if it requires propagation of information across calls and through paths in the procedures being called. An interprocedural problem is *flow insensitive* if it can be solved solely by propagation on call graphs, using summary information for each procedure[Cal88, Bur90].

their pointer aliasing calculation [CBC93]; it is difficult to compare our work to theirs, because they give no description of their algorithm.

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 eleven of the programs which appeared in [LR92] and one more. Measurements of average and maximum number of side effects found per assignment through dereference (i.e., **p=), per procedure and per call site have been made, as well as calculations of analysis times and the relative extra cost imposed by using our conditional analysis technique [LR91, LR92].

Most importantly, our results over the twelve programs show that on average 1.2 locations are assigned values per assignment statement through a dereferenced pointer variable (e.g., **p=), indicating that often there is only one alias for such a variable at a program point. Also, on average, less than 8% of all the visible variables at each such assignment 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.

This paper is organized as follows. Section 2 discusses our pointer aliasing algorithm and presents our decomposition of the modification side effects problem. Section 3 reports our empirical results in detail. Section 4 summarizes the contributions of this work. Appendix A presents a comparison of our MOD decomposition for C to that for FORTRAN. Appendix B gives an example of our analysis.

2 Flow Sensitive Interprocedural Analysis

2.1 Realizable Interprocedural Paths. 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[2]. 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. 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 by a call, is associated with data flow facts in the called procedure; it 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

[2] We do not allow *setjump* or *longjump* in programs analyzed.

at program point n is represented by an unordered pair ⟨x,y⟩ at n. Our encoding of the run-time stack is the set of *reaching aliases*[3] (*RA*s) that exists at entry of procedure p containing n when p is invoked. The *RA*s can be used to determine to which call sites, aliases at the exit of a called procedure should be propagated. In [LR92], we safely restricted the size of the reaching alias sets to one, yielding a compact and effective encoding of the run-time stack. 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]. Choi et. al. use the last call site encountered as their encoding of the run-time stack in their flow sensitive aliasing algorithm[CBC93, CB]. They also describe an algorithm variant that uses alias sets of unrestricted size, called *source alias sets*, as its encoding. We are jointly studying the precision and complexity effects of our two approaches and hope to compare algorithm performance in practice [MLR+93].

2.2 Pointer-induced Aliasing. Our MOD solution procedure requires the results of our pointer aliasing approximation algorithm. Therefore, in what follows, we give a brief overview of the algorithm, described in detail in [LR92].

Intraprocedurally, aliases induced by a reaching alias at a procedure entry, are associated with that reaching alias. Aliases that are created regardless of any reaching aliases, could legitimately be associated with any reaching alias, but for practicality, we only associate them with a special reaching alias, ϕ. We use $Calias(n, RA)$ to represent the set of aliases at program point n under the condition that the alias RA reaches the entry of the procedure containing n [LR92][4]. The intraprocedural propagation of aliases through pointer assignment statements is conceptually similar to the single level pointer aliasing algorithm in Chapter 10 of [ASU86] with extensions to handle multiple level pointers.

Interprocedurally, a call to procedure Q, $call_Q$, creates reaching aliases at the entry of Q. We use $Reach(call_Q, RA)$ to denote the set of reaching aliases induced by both the parameter bindings *and* the aliases associated with RA at the call (i.e., aliases in $Calias(call_Q, RA)$). The special reaching alias ϕ and reaching aliases created solely by the parameter bindings are included in the set $Reach(call_Q, \phi)$. At the exit of Q, aliases associated with reaching alias RA', are propagated to any call site $call_Q$, where $RA' \in Reach(call_Q, RA)$, and, thereafter are associated with RA in the procedure containing that call site. The actual algorithm includes details of name space mappings between the calling and called procedures [LR92]; for brevity, we omit them here.

2.3 Decomposition of the MOD problem. We are solving for modification side effects to fixed-locations at program points. *Fixed-locations* are either user-defined variables or heap storage creation site names/field accesses. For example, in C syntax **x** and **x.f** are fixed-locations whereas ***p** and **p->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, while we cannot distinguish between two fixed-locations created at the same site, we can distinguish those created at different sites. The side effects reported are differentiated by fixed-location type: global, local, dynamically-created, and non_visible (within that procedure). The *non_visibles* are local variables of other procedures or an earlier instantiation of the current procedure [LR92].

In solving for modification side effects, we decompose the MOD problem into subproblems that are individually easier to solve than the monolithic problem. We first solve the conditional aliasing problem

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

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

(i.e., *ALIAS*). 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* solutions can then be used to derive MOD information for program points, while *PMOD* solutions can be used to derive a procedure-level summary of modification side effects.

Our decomposition of the MOD problem is pictured in Figure 1, where P is a procedure, RA is a reaching alias and n is a program point. *ALIAS* is our solution to the conditional aliasing problem. *DIRMOD*(n) is the left hand side of the assignment at program point n. At an assignment n, *CondLMOD* widens *DIRMOD*(n) to include the effects of aliasing. *CondIMOD*(P, RA) summarizes *CondLMOD* information for each reaching alias RA over all assignment statements in procedure P. *PMOD* for P is formed from local *CondIMOD* information and *PMOD* information propagated from procedures called by P, thus calculating both direct and indirect side effects of P. *CMOD* at a call site is constructed from *PMOD* of the called procedure, and at an assignment, from *CondLMOD* of that statement. Finally, *MOD* at a statement is constructed from *CMOD* by summarizing over all reaching aliases. A comparison of our MOD decomposition for C to that for FORTRAN is given in Appendix A.

2.4 Data Flow Equations. In these discussions, we make several assumptions:

- *assignment* is synonymous with *value-setting statement*; Thus, **scanf** is considered an assignment.
- all variable names are unique; This allows us to ignore the issue of *name hiding* which can easily be accommodated by appending variable names with the function and file in which they are defined.
- call-by-value parameter passing as in C is used; Call-by-reference parameter passing can be transformed into call-by-value by adding an additional level of indirection[LRZ93].

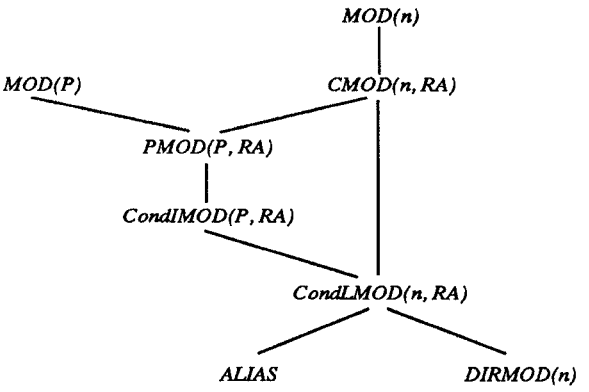

Figure 1: Decomposition of the MOD problem

- *on bottom* data flow information is computed (i.e., information at a statement incorporates the effects of that statement).
- our modification side effects sets are associated with our encoding of the run-time stack, reaching aliases, to restrict our attention to realizable paths; however, our MOD algorithm is independent of the choice of run-time stack abstraction.
- *Predecessors*(n) represents the set of predecessors of n in the program;
- trivial, reflexive aliases (e.g., $<*p,*p>$) are associated with the special reaching alias ϕ at all programs points; This assumption simplifies the equation for *CondLMOD*. In the actual implementation, we do not store these trivial aliases.

DIRMOD(n) was defined above as visible, direct side effects at a statement; therefore, it requires no data flow equation. *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 when RA reaches the entry of the procedure containing n. *CondLMOD* is specified by equation (1) in Figure 2. If *DIRMOD*(n) is a fixed-location, it is in *CondLMOD*(n,ϕ) because reflexive aliases are associated with the special reaching alias ϕ.

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

$$CondIMOD(P, RA) = \bigcup_{\substack{n \text{ an assign-} \\ \text{ment in } P}} CondLMOD(n, RA)$$

$$CondLMOD(n, RA) = \bigcup_{pred \in Predecessors(n)} \left\{ obj_1 \;\middle|\; \begin{array}{c} obj_2 = DIRMOD(n) \text{ and} \\ \langle obj_1, obj_2 \rangle \in Calias(pred, RA) \\ \text{and } obj_1 \text{ is a fixed-location} \end{array} \right\} \quad (1)$$

$$PMOD(P, RA) = CondIMOD(P, RA) \bigcup \bigcup_{\substack{call_Q \text{ in } P \text{ and} \\ RA' \in Reach(call_Q, RA)}} \left(b_{call_Q}(PMOD(Q, RA')) \right) \quad (2)$$

$$CMOD(n, RA) = \begin{cases} CondLMOD(n, RA) & \text{if } n \text{ is an assignment} \\ \bigcup_{RA' \in Reach(n, RA)} b_n(PMOD(Q, RA')) & \text{if } n \text{ is a call of Q} \\ \emptyset & \text{otherwise} \end{cases} \quad (3)$$

Figure 2: Data Flow Equations for *CondLMOD*, *PMOD* and *CMOD*

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 equation (2) in Figure 2, which can be solved iteratively. In the equation, $call_Q$ is a call site in P at which P calls Q. $Reach(call_Q, RA)$ represents the set of reaching aliases at the entry of Q induced by the parameter bindings at the call and aliases in $Calias(call_Q, RA)$. The function b_{call_Q}, specific to $call_Q$, maps names from the called procedure (Q) to the calling procedure (P) according to scoping rules [CK87] and only returns fixed-locations. Specifically, b_{call_Q} factors out all local variables of Q (including formal parameters of Q), maps global fixed-locations (global variables and dynamic storage locations) to themselves, and maps non_visibles in Q to their corresponding fixed-locations in P, which are either locals of P or non_visibles in P [LR92].

With the *PMOD* solutions, the modification side effects for calls and assignments are specified by equation (3) in Figure 2. Finally, *MOD*(n) summarizes the effects over all executions of n in procedure P and *MOD*(P) summarizes the effects over all calls of P. Both are obtained by considering all reaching aliases for P.

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

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

In Appendix B, we show *PMOD* and *CMOD* solutions for an example program.

2.5 Precision and Safety. The precision and safety of our MOD calculation depends upon the precision and safety of the underlying alias analysis. We address the issue of safety in [LRZ93], and address the issue of empirically measured precision in Section 3.

2.6 Worst-case complexity. We give a detailed analysis of the worst-case complexity of our MOD algorithm in [LRZ93]. In brief, given the following definitions:

- N_{alias} is the total number of conditional aliases in the program.
- N_{assign} is the number of assignments in the program.
- N_{fixed} is the number of fixed-locations.
- N_{ICFG} is number of nodes in our representation of the program. This is roughly equivalent to number of program points.
- N_{proc} is the number of procedures in the program.

- C_{union} is the cost of the union operation over sets of fixed-locations [$\Omega(N_{fixed})$].
- M_{call} is the maximum number of calls for any one procedure.
- M_{pred} is the maximum number of predecessors of any assignment.
- M_{RA} is the maximum number of reaching aliases at the entry of any procedure.

the worst-case time complexity for our MOD calculation is:

$$\mathcal{O}\left(\begin{array}{l} N_{proc} * M_{call} * M_{RA}^2 * N_{fixed}^2 + \\ N_{assign} * M_{RA} * M_{pred} * C_{union} + \\ N_{alias} + N_{ICFG} * M_{RA} * C_{union} \end{array}\right)$$

As for most static analyses, the worst-case time has little correlation with the observed behavior of the algorithm in practice. In the next section, we give some empirical timing results.

3 Empirical Results

We have implemented our MOD decomposition and have empirical results for eleven of the programs analyzed in [LR92] plus *compiler*, a compiler for a subset of Pascal. Our implementation is written in C and analyzes a reduced version of C that excludes: union types, casting[5], pointers to functions, exception handling, *setjump* and *longjump*. 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 programs we have analyzed and their sizes are in Figure 3. In this, and all subsequent figures, the programs are sorted by size in lines of code. We have separated out the assignments that are through a dereference (**thru-deref**), meaning the location assigned is determined by a pointer (e.g., the assignment ***p = 5;**). We have done this because these assignments have non-trivial MOD solutions, whereas other assignments (e.g., **i = 0;**) have trivial solutions. In Figure 3, we also present the time required on a *Sun Sparcstation 10* for the MOD calculation, the alias calculation, and a simple compile with *no* optimizations enabled. The reported MOD times *do not include* the alias times, so the total analysis time is the sum of these two columns. In all cases, the MOD times are less than that for a compile and, in most cases, are orders of magnitude smaller than the alias times. The total analysis time for the smaller programs is about the time of a compile, but for larger programs it is not. These results are encouraging, but we need to improve the efficiency of our alias analysis on larger programs.

In Figure 4, we give summary statistics for the MOD solution for **thru-deref** 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**: (non_visible) MOD information for local variables of *other* procedures or of an earlier recursive instantiation of the enclosing procedure. In our implementation [LR92], for efficiency we use one placeholder to represent all non_visible fixed-locations within a procedure.
- **tot**: MOD information for all fixed-locations.

We give three different summary statistics. **Average/assign** (**Maximum/assign**) is the average (maximum) number of fixed-locations modified by assignment statements. **Average percent/assign** 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[6] accessible through globals and formals at the entry of the enclosing procedure

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

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

program	lines of code	number of procedures	number of calls	number of assigns		MOD time (sec)	alias time (sec)	compile time (sec)
				all	thru-deref			
allroots	188	8	20	96	33	0.01	0.23	0.79
diffh	268	16	51	117	19	0.09	0.78	1.54
fixoutput	458	8	14	133	90	0.05	0.39	0.79
ul	541	19	73	262	42	0.20	2.70	1.29
lex315	776	19	104	179	54	0.18	1.05	1.29
pokerd	1130	28	87	384	104	0.32	13.87	2.59
loader	1539	33	86	330	119	0.38	15.13	2.75
diff	1782	45	166	764	232	0.86	12.74	8.48
football	2354	61	265	1070	267	0.88	3.78	8.92
compiler	2360	40	363	373	72	0.82	1.38	4.17
assembler	3361	55	256	691	290	1.72	84.54	5.63
simulator	4663	102	413	872	274	0.88	20.72	7.94

Figure 3: Program size and analysis time

program	Average/assign					Average percent/assign					Maximum/assign				
	glo	dyn	loc	nv	tot	glo	dyn	loc	nv	tot	glo	dyn	loc	nv	tot
allroots	0.9	0.1	0.0	0.0	1.0	13%	9%	0%	0%	8%	1	1	1	0	1
diffh	0.7	0.2	0.2	0.0	1.1	4%	16%	3%	0%	5%	1	1	1	0	2
fixoutput	0.8	0.1	0.1	0.0	1.0	6%	4%	3%	0%	5%	1	2	1	0	2
ul	0.6	0.0	0.4	0.0	1.0	2%	0%	11%	0%	2%	1	0	1	0	1
lex315	0.8	0.3	0.0	0.0	1.1	6%	9%	0%	0%	6%	1	2	1	0	2
pokerd	0.5	0.2	0.2	0.2	1.1	2%	1%	4%	8%	2%	1	3	1	3	3
loader	0.6	0.2	0.0	0.5	1.4	3%	1%	<1%	10%	3%	1	2	1	9	9
diff	0.7	0.3	0.1	0.0	1.1	1%	1%	1%	0%	1%	2	2	1	0	2
football	1.0	0.0	0.0	0.0	1.0	1%	0%	<1%	<1%	1%	3	0	1	1	3
compiler	1.0	0.0	0.0	0.0	1.0	2%	0%	1%	0%	2%	1	0	1	0	1
assembler	0.6	0.2	0.1	0.6	1.4	2%	<1%	<1%	11%	2%	2	1	2	9	9
simulator	0.6	0.2	0.0	0.6	1.4	2%	1%	<1%	15%	2%	1	2	1	13	13

Figure 4: MOD statistics for thru-deref assignment statements

Then **percent/assign** is simply the number of fixed-locations modified, divided by the number of potentially modified locations of the appropriate type per assignment. The **average percent/assign** is the average of **percent/assign** over all assignments. For some assignments the number of possible locals and the number of non_visibles are zero; in these cases, we use "0%" as the **percent/assign**.

The results in Figure 4 are extremely encouraging. Any executable assignment in a normally terminating program will modify at least one fixed-location. Thus, one is a lower bound of **tot** for **average/assign**. The values in **tot** column of **average/assign** are all close to one with a maximum value of 1.4. This indicates that our algorithm is highly precise. In this Section 3.1, we present additional empirical evidence on the precision of our calculation. **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 8% indicating that our MOD calculation is yielding far more accuracy than the worst-case assumption and therefore is worth performing. **Maximum/assign** is interesting, but cannot

easily be used to justify the quality of our calculation. We have investigated the high values in **tot** column of **maximum/assign** for loader, assembler, and simulator by hand checking the solutions at the assignment that generated the maximum. Our algorithm found no spurious modifications for these statements.

Figure 5 and Figure 6 have the same structure as Figures 4 and are also encouraging; however, it is harder to get a good lower bound on how many fixed-locations are modified in these cases. We think the numbers reported are surprisingly small. In Figure 5, modified locals of the *called* procedure are not counted in the totals for a call site as those locations do not exist before nor after the call. 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% in Figure 6. We do not see this value because some procedures do not have any locals; these procedures introduce a 0% into the average calculation. The high values in **tot** column of **maximum/procedure** in Figure 6 are expected because procedure *main* of each program will directly or indirectly modify every location in the program, except locals of other procedures.

3.1 Empirically Measured Precision. We have empirically bounded the precision of our calculation in a similar manner as we did in [LR92]. There we explained the two sources of imprecision in the aliasing calculation: *k-limiting,* resulting from the necessity for approximation to handle *a priori* unboundable dynamic data structures and *control flow,* resulting from safe assumptions about the actual execution paths with which aliases are associated. Our empirical measurements give a worst-case estimate of the latter type of approximation; there is no viable way of measuring the former. All pointer aliasing algorithms must use some k-limiting approximation.

We associate with each fixed-location in the MOD solution either *yes* or *maybe* with the following interpretation:

program	MAYBE at assigns	MAYBE at calls	MAYBE at procs
allroots	0%	0%	0%
diffh	0%	0%	0%
fixoutput	0%	0%	0%
ul	0%	0%	0%
lex315	0%	0%	0%
pokerd	<1%	2%	2%
loader	<1%	2%	1%
diff	<1%	<1%	1%
football	0%	0%	0%
compiler	0%	0%	0%
assembler	3%	2%	3%
simulator	3%	15%	9%

Figure 7: Percent *maybes* in the MOD solutions

- (a,*yes*) ∈ MOD(X) implies that X definitely modifies a on some execution.
- (a,*maybe*) ∈ MOD(X) implies that X may or may not modify a, but for safety we assume X modifies a.

The details of how to compute the MOD solutions with this added requirement are in [LRZ93]. We give the percentage of *maybes* in our MOD solutions in Figure 7. The percentage of fixed-locations spuriously reported as modified due to control-flow approximations for all assignments, calls, and procedures, can be at most the percentage of *maybes* reported.

Finally, we have empirical evidence that by associating reaching aliases (RA) with *CMOD* and *PMOD*, we are not incurring much unnecessary work in our algorithm[LRZ93]. This would be possible, if some assignment (or call or procedure) modifies the same fixed-location under many different reaching aliases. 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 computed the ratio of the *CMOD* and *PMOD* solution size to the MOD solution size. For the twelve programs in this paper, the maximum ratio is 1.04, which indicates that little redundant work is being performed. The ratios for these programs appear in [LRZ93].

program	Average/call					Average percent/call					Maximum/call				
	glo	dyn	loc	nv	tot	glo	dyn	loc	nv	tot	glo	dyn	loc	nv	tot
allroots	0.9	0.2	0.0	0.0	1.2	14%	25%	0%	0%	11%	2	1	0	0	3
diffh	2.4	0.5	0.0	0.0	2.9	15%	47%	0%	0%	14%	9	1	0	0	10
fixoutput	6.1	2.5	0.0	0.0	8.6	43%	83%	0%	0%	46%	10	3	0	0	13
ul	2.5	0.0	0.0	0.0	2.5	6%	0%	0%	0%	6%	31	0	0	0	31
lex315	3.4	1.7	0.0	0.0	5.1	22%	58%	0%	0%	28%	9	3	0	0	12
pokerd	1.6	0.6	0.2	0.0	2.4	5%	5%	2%	0%	5%	23	13	1	0	36
loader	1.2	1.6	0.6	0.1	3.5	7%	8%	13%	4%	8%	11	19	3	1	30
diff	2.3	1.3	0.0	0.0	3.6	3%	4%	0%	0%	3%	58	36	0	0	94
football	3.6	0.0	0.0	0.0	3.6	5%	0%	<1%	0%	4%	67	0	1	0	67
compiler	12.8	0.0	0.0	0.0	12.8	25%	0%	0%	0%	24%	45	0	0	0	45
assembler	2.6	2.1	0.3	0.2	5.2	8%	9%	6%	5%	8%	24	24	5	5	48
simulator	1.7	0.4	0.2	0.1	2.4	5%	3%	7%	5%	4%	24	16	2	3	40

Figure 5: MOD statistics for procedure calls

program	Average/procedure					Average percent/procedure					Maximum/procedure				
	glo	dyn	loc	nv	tot	glo	dyn	loc	nv	tot	glo	dyn	loc	nv	tot
allroots	2.2	0.6	2.8	0.0	5.6	32%	63%	75%	0%	52%	7	1	6	0	8
diffh	3.6	0.5	1.8	0.0	5.9	23%	50%	50%	0%	30%	16	1	7	0	17
fixoutput	7.0	2.4	0.8	0.0	10.2	50%	79%	25%	0%	56%	14	3	4	0	17
ul	8.9	0.0	1.2	0.0	10.1	22%	0%	42%	0%	24%	40	0	5	0	40
lex315	4.6	1.7	1.5	0.0	7.9	31%	58%	26%	0%	40%	15	3	16	0	18
pokerd	4.7	2.3	3.2	0.4	10.6	15%	18 %	79%	38%	22%	31	13	12	3	44
loader	2.7	3.7	2.5	1.0	9.9	16%	20%	58%	74%	23%	17	19	18	9	48
diff	7.1	4.5	3.0	0.0	14.5	10%	12%	76%	0%	13%	71	36	11	0	107
football	7.4	0.0	2.9	0.0	10.3	9%	0%	69%	10%	12%	81	0	36	1	81
compiler	24.2	0.0	1.1	0.0	25.2	47%	0%	63%	0%	48%	52	0	4	0	52
assembler	6.3	5.6	2.7	1.6	16.3	19%	23%	69%	48%	25%	33	24	13	15	60
simulator	3.6	1.4	1.9	0.8	7.7	10%	9%	79%	81%	14%	36	16	7	24	52

Figure 6: MOD statistics for procedures

4 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.

Acknowledgments We thank our colleagues at Siemens Corporate Research for their help with the front end, *ptt*. We also thank Don Smith and the referees for insightful comments on this paper.

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 dependence 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.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, June 1988.

[CB] Jong-Deok Choi and Michael Burke. personal communication.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[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.

[LRZ93] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. Technical Report LCSR-TR-201, Laboratory for Computer Science Research Technical Report, March 1993. This report supersedes LCSR-TR-195 and is an expansion of our ACM SIGPLAN PLDI'93 paper.

[MLR+93] Thomas J. Marlowe, William Landi, Barbara G. Ryder, Jong-Deok Choi, Michael Burke, and Paul Carini. A cost-precision comparison of two flow sensitive interprocedural algorithms for pointer-induced aliasing. Technical report, Laboratory for Computer Science Research Technical Report, March 1993. in preparation.

[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. being revised for journal 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.

MOD Decomposition for FORTRAN[CK87] s is a statement. P is a procedure.	**MOD Decomposition for C** n is either an assignment or a call. P is a procedure. RA is a reaching alias.
$LMOD(s)$ the set of variables modified by an execution of s, excluding any procedure calls in s	$CondLMOD(n, RA)$ n is an assignment. the set of fixed-locations modified by an execution of n considering aliases that are associated with RA and hold on entry n.
$IMOD(P)$ the set of variables modified by an invocation of P, excluding any procedure calls in P	$CondIMOD(P, RA)$ the set of fixed-locations modified by an invocation of P, considering only assignments in P and aliases associated with RA in P
$IMOD^+(P)$ the set of variables either modified directly in P or modified as reference formals in procedures called in P	$CondIMOD^+(P, RA)$(see[LRZ93]) the set of fixed-locations either modified directly in P or modified as non_visibles in procedures called by P, considering only aliases associated with RA in P
$GMOD(P)$ the set of variables modified by an invocation of P, including procedure calls in P and ignoring any aliases in P	$PMOD(P, RA)$ the set of fixed-locations modified by an invocation of P, considering both assignments and procedure calls in P, and aliases associated with RA in P
$DMOD(s)$ the set of variables modified by an execution of s, including procedure calls in s and ignoring any aliases in the procedure containing s	$CMOD(n, RA)$ n is either an assignment or a call. the set of fixed-locations modified by an execution of n, considering aliases that are associated with RA and hold on entry n, and parameter bindings if n is a call
$MOD(s)$ the set of variables modified by an an execution of s, considering all aliases in the procedure containing s	$MOD(n)$ n is either an assignment or a call. the set of fixed-locations modified by an execution of n, considering all possible aliases true on entry n in the procedure containing n

Figure 8: Comparison of MOD decompositions for FORTRAN and C

A Comparison with the MOD Decomposition for FORTRAN

Our decomposition of the MOD problem for C is similar in structure to the original decomposition for FORTRAN by Banning [Ban79], in the sense that both calculate local side effects in each procedure first, and then set up data flow equations on call graphs to compute procedure-level side effects (i.e., a flow insensitive interprocedural calculation).

The two decompositions are also similar in what is included in the MOD sets. In FORTRAN programs, variables are the only fixed-locations and therefore various MOD sets in the decomposition for FORTRAN include just variable names. In C, pointer variables and dynamic allocation are allowed. Although simple variable names (e.g., **p**) still represent fixed-locations, names with dereferences (e.g., ***p**) can potentially denote different locations during execution and thus are not considered fixed-locations. We handle dynamic allocations by naming each site so that dynamically allocated locations are identified by the sites creating them. The MOD sets in our decomposition include variable names and names for dynamic allocation sites.

The two decompositions differ in their treatment of aliases. In the FORTRAN decomposition, aliases are computed at procedure calls. This is possible because for FORTRAN programs, only procedure calls can create aliases and aliases created by a call hold throughout execution of the procedure being called. In our MOD decomposition for C, aliases are computed at pointer assignments and procedure calls, because aliases vary intraprocedurally. An alias at a program point is associated with a reaching alias for the procedure containing that program point. These reaching aliases differentiate side effects caused by different calls of the same procedure.

In Figure 8, we compare various MOD sets defined in our MOD decomposition for C and those in the decomposition for FORTRAN as presented in [CK87].

B An Example

We show the results of our analysis for the example

```
int x, y, k;
R(b)
int *b;
{
  if (*b)
    { b = &k;
      *b = 0; }
  *b++;
}
main()
{
  R(&x);
  R(&y);
}
```

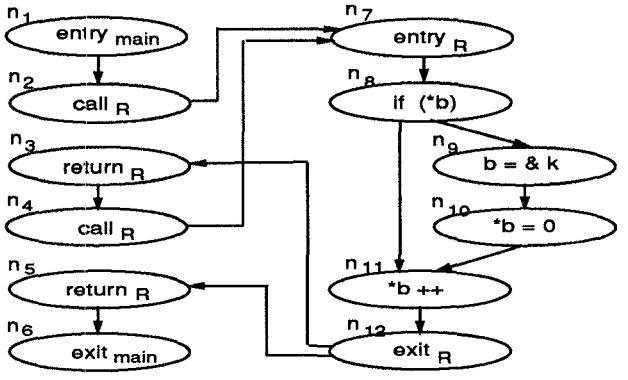

Figure 9: An example program and its ICFG

Reaching Alias	Alias Solutions for 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>

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 }

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 10: Aliases, *PMOD* and *CMOD* solutions for the example program

program in Figure 9. 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*. The alias solution for R is shown in Figure 10. The *PMOD* and *CMOD* solutions computed according to our decomposition are also shown in the same figure. Empty entries in these tables mean either no alias or no side effect.