Comparing Flow and Context Sensitivity on the Modification-side-effects Problem

Philip A. Stocks[†] **Barbara G. Ryder**[†] **William A. Landi**[‡] **Sean Zhang**[†]

[†]Dept of Computer Science
Hill Center, Busch Campus
Rutgers University
Piscataway, NJ 08855, USA
{pstocks,ryder,xxzhang}@cs.rutgers.edu

[‡]Siemens Corporate Research
755 College Road East
Princeton, NJ 08540, USA
landi@scr.siemens.com

ABSTRACT

Precision and scalability are two desirable, yet often conflicting, features of data-flow analyses. This paper reports on a case study of the modification-side-effects problem for C in the presence of pointers from the perspective of contrasting the flow and context sensitivity of the solution procedure with respect to precision and scalability. The results show that the cost of precision of flow- and context-sensitive analysis is not always prohibitive, and that the precision of flow- and context-insensitive analysis is substantially better than worst-case estimates and can be sufficient for certain applications. Program characteristics that affect the performance of data-flow analysis are identified.

Keywords

Interprocedural data-flow analysis, modification side effects, flow sensitivity, context sensitivity, empirical study, pointer aliasing.

INTRODUCTION

Accurate compile-time calculation of possible interprocedural side effects is crucial for aggressive compiler optimization, practical dependence analysis in programs with procedure calls, data-flow testing, incremental semantic change analysis of software, program understanding tools, and effective static interprocedural program slicing. Many of these key applications in parallel and sequential programming environments need interprocedural def-use information which can be approximated using side-effect information. The utility of tools to solve these problems is directly dependent on the accuracy of the data-flow information available to them. These applications need an efficient method to report program-point-specific side-effect information in the presence of pointers in order to handle modern languages such as C, C^{++}, Fortran90 and Java; this requires practical interprocedural side-effect analysis with pointers, something that previous techniques for FORTRAN cannot supply [1, 4, 5, 2].

A family of side-effect analyses for C, parameterized by the pointer aliasing algorithm used, has been defined [11, 10]; these analyses vary in the cost/precision tradeoffs for the calculated side-effect information. Using different combinations of flow-sensitive and/or context-sensitive analyses, it is possible to offer a range of cost/precision levels for different applications. This paper reports on the *first comparative experiments* on the effectiveness of flow and context sensitivity in data-flow analyses as measured on interprocedural side-effect analysis of C codes (i.e., MOD$_C$). The empirical experiments reported involve two MOD$_C$ methods [10] with different component pointer aliasing algorithms. MOD$_C$(FSAlias) uses a flow-sensitive, calling-context-sensitive approximation algorithm for pointer-induced aliasing [9]. MOD$_C$(FIAlias) uses a flow-insensitive, calling-context-insensitive approximation algorithm for pointer-induced aliasing which is similar to the algorithm described in [21].[1] MOD$_C$(FSAlias) reports program-point-specific possible modification side effects; the results are more precise than information derivable using the same alias summary for all statements of a procedure. MOD$_C$(FIAlias) also reports program-point-specific possible modification side effects, but it uses alias information that is assumed to be valid globally throughout the program; thus, more spurious side effects may be reported.

These are the *first* MOD$_C$ algorithms with extensive implementation results reported. Surprisingly, these show that the precision of flow- and context-sensitive analysis is obtainable practically and that the lower precision of flow- and context-insensitive analysis is sufficient for some applications. It is important that the effectiveness of the analysis was profiled with respect to a specific application, namely, program understanding, in which a user is interested in indirect side effects through a dereferenced pointer. Measurement for a specific application more accurately depicts the utility of the data-flow solution obtained than any metrics on the data-flow solution alone.

The empirical tests of these algorithms used 45 C programs, most of which are publicly available. Measurements of average number of side effects found per assignment through

---

[1] The implemented algorithm handles unions and casting in C programs which the published version of the algorithm does not.

dereference (i.e., a *through-dereference* assignment statement such as `*p=`) and per call site have been recorded for both algorithms. Significantly better precision is obtained by MOD$_C$(FSAlias) at greater time cost than MOD$_C$(FIAlias). This precision is necessary for some compiler transformations, where MOD$_C$ information is used to approximate def-use associations. MOD$_C$(FSAlias) shows surprising scalability on programs up to 10,000 lines of code at cost similar to compile-time cost in the prototype; larger programs which use a specific programming style (i.e., little use of recursive structures) also can be handled. Unexpectedly, MOD$_C$(FIAlias) is much more accurate than a coarse worst-case estimate and usually costs an order of magnitude less[2] than MOD$_C$(FSAlias), so it may be sufficient and practical for program understanding applications on large codes. A crude measure of the accuracy of MOD$_C$(FIAlias) versus MOD$_C$(FSAlias) can be obtained by examination of the difference in their solutions on the data admitting both kinds of analysis, since both are *safe* estimates of side effects that can occur. Normalized differences at through-dereference statements and calls are presented in the empirical section.

In summary, the empirical results show the utility of both analyses for specific applications and demonstrate the precision gains from sensitivity in the aliases and thus in the side-effect information obtained. Recent work in partitioning programs for analyses [21, 17] yields hope that analyses of varying cost and precision can be applied to different parts of a program to obtain desired data-flow information at practical cost.

BASIC CONCEPTS

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* of a data-flow problem [13]. For interprocedural data-flow analysis, not all paths in the usual 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 [7, 9, 15, 20]. 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.[3]

Data-flow algorithms which obtain differentiated program-point-specific information are typically *flow-sensitive*; that is, they propagate information across calls, along paths in the called procedure and then back again into the calling procedure. In contrast, an algorithm is *flow-insensitive* if it propagates information solely on call multigraphs, using summary information for each procedure which can be gathered without exploring paths in that procedure [14]. In order to restrict propagation of data-flow information to realizable paths, an algorithm needs to "match up" calls with corresponding returns when passing information from a called routine back to its caller. If data-flow information is kept separate by calling context, so that potentially different information is returned to different call sites, the algorithm is termed *context-sensitive*; otherwise, the same information is returned to all calling contexts and the algorithm is termed *context-insensitive*.

The calling context approximation used in the MOD$_C$ schema is the same as that of the $FSAlias$ algorithm in [8, 9]. 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. The encoding of calling context is the set of *reaching aliases* (RAs)[4] that exists at entry of procedure p containing n when p is invoked from a particular call site. The RA set is used to determine to which call sites aliases at the exit of a called procedure should be propagated, namely only to those call sites that induce that RA set at procedure entry. Using each single alias pair from the RA set one at a time yields a safe approximate solution for multiple levels of dereferencing; this is the approximation used for calling context in MOD$_C$(FSAlias).[5] The empirical results show it is a good approximation in practice.

A program is represented by a common directed graph structure, an ICFG or *interprocedural control-flow graph*. This is no more than the control-flow graphs of each procedure connected together at call sites, each of which has been split into a call and return node. Each procedure is presumed to have a single entry and single exit node, which is easy to ensure by insertion of extra nodes/edges. Each call node is connected to the called procedure's entry node; each return node is connected to the called procedure's exit node.

Modification side effects are reported for fixed locations at program points. *Fixed locations* are either user-defined variables or heap storage creation site names/field accesses. For example in C, `x` and `x.f` are fixed locations whereas $*$`p` and `p->f` are not. Each dynamically allocated fixed location is identified by the program point that created it, a common approach. Therefore, while two fixed locations created at the same allocation site are not distinguishable, those created at different sites are distinguishable.

All data-flow algorithms must deal with the *a priori* unbounded nature of recursive data structures. The $FSAlias$ algorithm uses Jones and Muchnick's *k-limiting* definition for recursive data structures, which truncates names to at most k distinct field references [6]. The $FIAlias$ algorithm only computes aliases for names appearing in the program context and thus, has no need for k-limiting.

Aggregate data structures are handled specially. Arrays are treated as a single variable with the assumption that pointer arithmetic stays within array bounds. Side effects to the independent fields of a structure are distinguished.

---

[2] The decreased cost is primarily due to the cost of the $FIAlias$ phase relative to $FSAlias$.

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

[4] Reaching aliases were referred to by the term *assumed aliases* in [9].

[5] For aliasing in programs restricted to one level of dereferencing, the RA sets are of cardinality one and can be used to obtain a precise solution [8].

MOD$_C$ SCHEMA

The MOD$_C$ schema defines a family of algorithms which solve for modification side effects to fixed locations at program points. Side effects reported are differentiated by fixed-location type: *global, local, dynamically-created,* and *non-visible*.[6] In solving for modification side effects, the MOD$_C$ problem is decomposed into subproblems that are individually easier to solve than the monolithic problem. The problem decomposition assumes that context-sensitive alias information is available; it preserves calling context with the side-effect information for as long as possible in the solution process. If the pointer alias algorithm used is context-insensitive, then conceptually all calling contexts are mapped to one context; that is, there is no differentiation in side effects returned from a procedure to any individual call site. The MOD$_C$ schema is described for the context-sensitive case; for a context-insensitive algorithm, each of the multiple subproblems distinguished by calling context becomes a single subproblem.

The first pass of the algorithm solves for pointer aliasing information, $ALIAS(n, RA)$. The subsequent analysis steps comprise the second pass. Given the results of this analysis, two related problems are calculated (i.) *PMOD*, a procedure-level summary of context-sensitive modification side-effects which can occur, and (ii.) *CMOD*, a set of modified fixed locations at each program point corresponding to a specific context. *CMOD* solutions can then be used to derive side-effect information for program points ($MOD(n)$), while *PMOD* solutions can be used to derive a procedure-level summary of modification side effects ($MOD(P)$).

The decomposition of the MOD$_C$ problem is pictured in Figure 1, where P is a procedure, RA is a calling context (i.e., for MOD$_C$(FSAlias), RA is a reaching alias) and n is a program point. A full description of each subproblem and its corresponding data-flow equations is available in [11, 10]. $ALIAS(n, RA)$ is the pointer alias solution. *DIRMOD*(n) captures all variable expressions which occur on the left hand side of the assignment at program point n (e.g., `*p=, v=`). At an assignment n, *CondLMOD* widens *DIRMOD*(n) to include the effects of aliasing. *CondIMOD*(P, RA) summarizes *CondLMOD* information for each calling context 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 contexts.

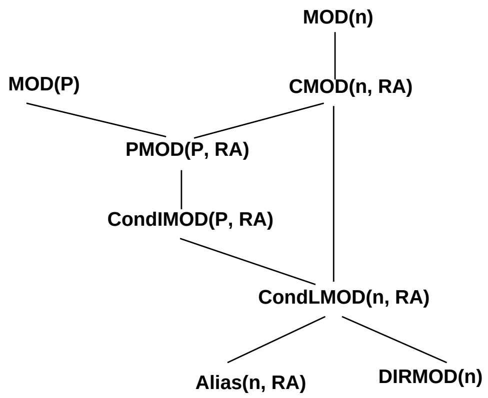


Figure 1: Decomposition of the MOD problem

EMPIRICAL RESULTS

This section describes and discusses execution results of the MOD$_C$ analyses. The MOD$_C$, $FSAlias$, and $FIAlias$ analysis code is implemented in C and analyzes a reduced version of C that excludes pointers to functions, exception handling, *setjump* and *longjump*. The results were gathered on a *Sun Sparcstation 20* with 348 Mb of RAM.

Table 1 shows information about the 45 C programs that were analyzed. The programs are ordered by the number of ICFG nodes; this order is maintained in subsequent figures. The numbers of procedure calls, call statements, and assignment statements in each data program are shown. For $MOD(n)$, the relevant statements in a program are assignments and procedure calls. Assignments through a pointer dereference are distinguished because these assignments have non-trivial solutions, whereas other assignments (e.g., `i = 0;`) have trivial solutions.

The last column of Table 1 indicates whether or not the $FSAlias$ succeeds in calculating an alias solution for the program. $FSAlias$ is unable to calculate a solution for 8 of the programs, because it runs out of space; $FIAlias$ can calculate a solution for all the programs.

Table 1 raises the question of what characteristics of a program affect the availability of a flow- and context-sensitive alias solution. Certainly program size is a factor because of the relationship between size of solution and size of program, but it can't be the only factor as the contrasting results for *moria* and *zip* show. *Moria* is a "large" program with a $FSAlias$ solution. Nor do the size differences between *zip* and *flex* seem vast enough to indicate a threshold on the power of $FSAlias$.

The root of the problem is caused by recursive data structures. The $FSAlias$ algorithm uses the somewhat naive *k-limiting* approximation to handle recursive data structures. However, in the cases of these larger programs that make excessive use of recursive data structures, the analysis gets bogged down in the generation and propagation of k-limited aliases. *Mo-*

---

[6] Non-visibles in procedure p are locations which map to no identifier in the scope of p; for example, a non-visible may be a local variable of the calling procedure which is accessible in the called procedure through an alias [9, 10].

Program	LOC	ICFG Nodes	# Procs	# Calls	# Assigns		Flow-sens. Alias Soln
					All	Thru-deref	
allroots	215	422	8	20	72	3	✓
fixoutput	401	617	7	13	123	5	✓
diffh	1726	646	15	50	80	4	✓
travel	862	698	16	24	170	3	✓
ul	548	1027	15	36	168	6	✓
plot2fig	1495	1077	27	78	159	16	✓
lex315	719	1297	18	103	137	6	✓
compress	1490	1319	16	29	274	11	✓
clinpack	1226	1429	14	80	267	30	✓
loader	1219	1563	31	80	242	78	✓
mway	705	1576	22	43	406	71	✓
ansitape	1596	1747	36	110	274	21	✓
stanford	887	1771	48	80	369	42	✓
pokerd	1243	1895	27	86	296	59	✓
zipship	1283	1955	14	53	332	59	✓
dixie	2109	2341	36	83	394	73	✓
zipnote	3155	2407	20	71	348	86	✓
learn	1483	2626	36	80	432	59	✓
xmodem	1712	2672	28	156	447	97	✓
compiler	2232	3008	39	350	304	2	✓
zipcloak	3644	3033	30	93	424	104	✓
sim	1439	3034	17	29	818	130	✓
cdecl	1015	3196	33	204	448	25	✓
diff	1726	3300	43	129	569	100	✓
unzip	4106	3416	40	99	731	52	✓
assembler	2673	3601	53	248	533	233	✓
gnugo	2901	3651	29	89	650	109	✓
livc	1886	4101	87	204	885	243	✓
lharc	3303	4250	87	198	791	123	✓
patch	2672	4608	56	271	750	135	✓
simulator	3733	5574	100	409	666	107	✓
arc	7507	5856	96	237	1105	160	✓
triangle	1930	6119	19	43	1072	241	✓
tbl	2511	6162	85	316	907	279	✓
football	2222	7313	59	258	847	225	✓
flex	6970	7376	86	307	1505	248	✓
zip	7427	9288	110	324	1554	331	
072.sc	8087	13690	154	698	1826	201	
spim	19032	16740	171	984	1587	383	
larn	9546	21184	270	2309	2536	158	
tsl	14646	27302	450	2109	2350	587	
008.espresso	13567	30510	319	1909	5125	1546	
moria	24596	38572	449	3708	5893	1493	✓
TWMC	23833	51627	210	844	11175	3957	
nethack	28735	58317	489	2902	4335	923	

Table 1: Program data set

ria has no recursive data structures. *Zip* uses recursive data structures much more heavily than *flex*.

MOD_C precision is reported in terms of the average number of fixed locations reported modified per kind of statement. Structure assignments raise the issue of how to count modifications to structure fields. The MOD_C analysis counts an assignment to a structure, say with three fields, as one fixed-location modification, and not as three. Counting is more complex when considering the total locations modified by a procedure; details can be found in [10], but are not required for the data presented here.

Precision at Through-dereference Statements

Figures 2(a) and 2(b) report the average numbers of fixed locations modified per assignment through a pointer dereference for both MOD_C(FSAlias) and MOD_C(FIAlias). The MOD_C(FIAlias) result for *moria* is elided because it skews the figure; the raw numbers are presented instead. The bars for each program are divided into the kind of location being modified. Each segment of the average bar is the average over the numbers in Figure 2 for that kind of location being modified. Notice that the average number of fixed locations modified is not closely correlated to program size.

How precise are these results? Any executable assignment in a normally terminating program will modify at least one fixed location. Thus, 1 is a lower bound of total fixed locations modified per assignment statement (the dotted lines in Figures 2(a) and 2(b) show the line $x = 1$). The precision of these results for MOD_C(FSAlias) is very encouraging, and highlights the precision of the flow- and context-sensitive aliasing. The totals are all close to 1 with a maximum value of 2 and an average of 1.3. In contrast, MOD_C(FIAlias) is more imprecise, averaging 5.1, and sometimes being wildly inaccurate as in such cases as *moria*[7]. *Moria* is a large program with very many large (though non-recursive) data structures with several aliases. Perhaps the inability of the MOD_C(FIAlias) algorithm to distinguish calling context and its inability to kill aliases explains the massive distortion between the MOD_C(FSAlias) and MOD_C(FIAlias) solutions for *moria*.

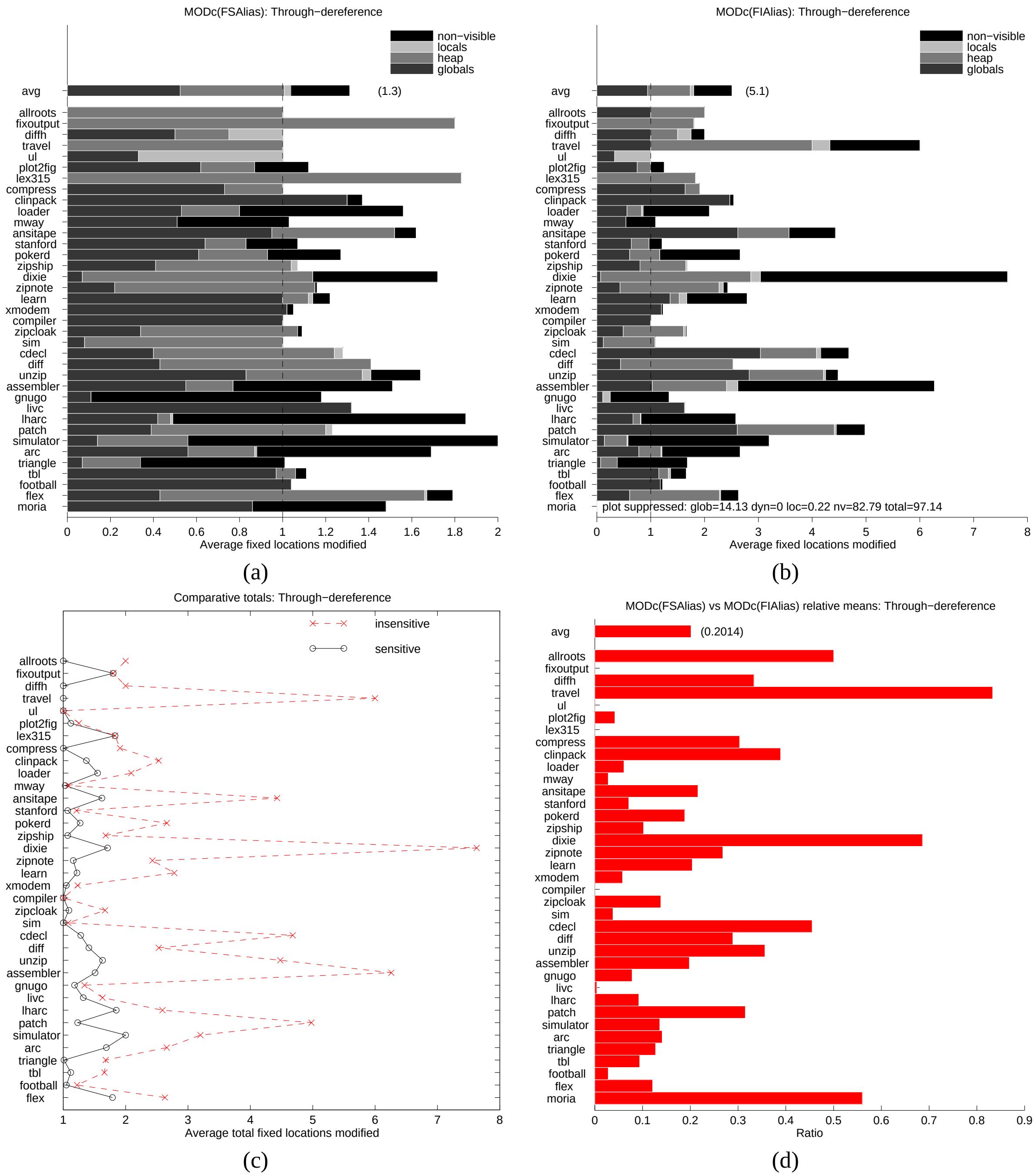
Figure 2(c) compares the average totals from Figures 2(a) and 2(b) in a more visually apparent manner. Again, *moria* is excluded from this comparison since it skews the figure. Finally, Figure 2(d) addresses the comparative difference between the MOD_C(FSAlias) and MOD_C(FIAlias) solutions. If $sens$ is the number of fixed locations reported modified at some point by MOD_C(FSAlias), and $insens$ is the number reported by MOD_C(FIAlias) at that point, then Figure 2(d) shows the average of the calculation $(insens - sens)/insens$ over each through-dereference assignment in the program. This measurement indicates the proportion of the MOD_C(FIAlias) solution that must be in error. Low numbers here mean that the MOD_C(FIAlias) solution is nearly as precise as the MOD_C(FSAlias) solution. Zero means that the solutions are the same.

Despite these results, the precision of the MOD_C(FIAlias) analysis is not to be understated. Figure 3 shows the average proportion of reported fixed locations modified at through-dereference assignments to the number of fixed locations potentially modified. The number of fixed locations potentially modified by an assignment is the sum of the number of globals in the program, the number of dynamic allocation sites in the program, the number of locals in the enclosing procedure, and the number of accessible non-visibles. Figure 3 shows what percentage the average totals reported by MOD_C(FIAlias) are of this worst-case. In parentheses after each bar is the average total number of fixed locations reported modified. Very low percentages indicate that the worst-case MOD_C solution is very much larger than what can be calculated using even flow- and context-insensitive data flow.

Tables 2 and 3 present another view comparing MOD_C(FSAlias) and MOD_C(FIAlias) that show histograms of the numbers of statements that modify a certain number of fixed locations. All the through-dereference assignments in all the programs are considered together when constructing these histograms[8]. The histograms are

[7] Without *moria* the average total for MOD_C(FIAlias) is 2.6.

[8] The programs with no $FSAlias$ solution are omitted when calculating


MODc(FSAlias): Through−dereference

MODc(FIAlias): Through−dereference

Comparative totals: Through−dereference

MODc(FSAlias) vs MODc(FIAlias) relative means: Through−dereference

(a) (b)

(c) (d)

Figure 2: Fixed locations modified by through-dereference assignments

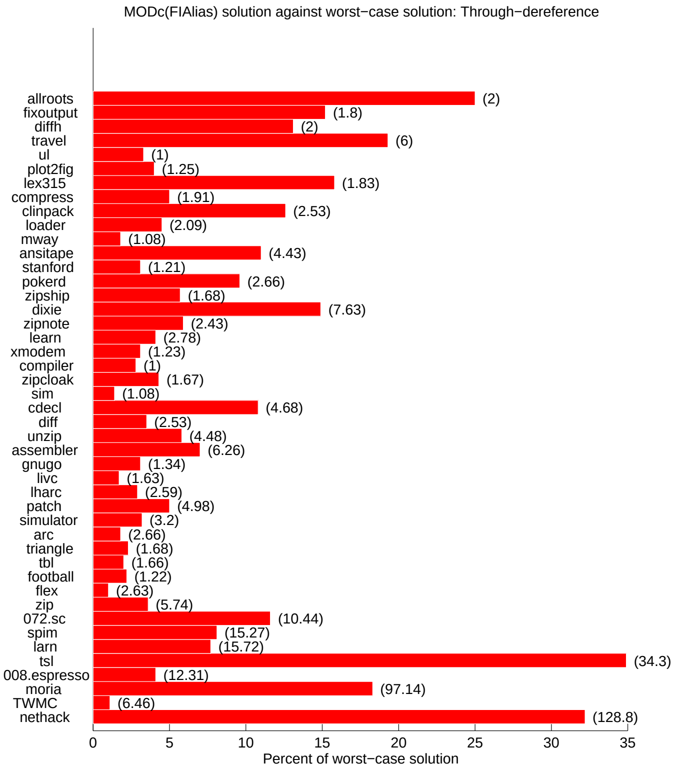


Figure 3: Through-dereference MOD$_C$(FIAlias) solution as percentage of worst-case solution.

broken down by fixed-location kind. For example, Table 2 says that 1803 of the through-dereference assignments modify 0 globals. The percent below column shows what percentage of the through-dereference assignments have that many total side effects or more[9]. Gaps in the sequence indicate that there are no statements with that many side effects.

Again, the fact that 96% of the assignments report modifications to only 1 location in the MOD$_C$(FSAlias) analysis is indication of its precision. Contrast that to the MOD$_C$(FIAlias) histogram where 50% of the assignments are reported as modifying 3 or more fixed locations.

For completeness, Figure 4 shows the results of MOD$_C$(FIAlias) on the programs for which MOD$_C$(FSAlias) does not generate a solution. Though in some cases these averages are high, they still represent vast improvement over the worst case, as shown by Figure 3.

For brevity, only assignments through a pointer dereference

# Side Effects	Glob	Dyn	Loc	Nv	Total	Percent Below
0	1803	3766	4717	3871	7	100.00
1	2857	708	21	592	4103	99.85
2	43	240	0	82	363	13.25
3	27	10	0	56	96	5.59
4	0	2	0	43	49	3.57
5	2	0	0	13	20	2.53
6	0	10	0	23	24	2.11
7	1	2	0	1	14	1.60
8	3	0	0	12	15	1.31
9	0	0	0	5	5	0.99
10	0	0	0	10	1	0.89
11	0	0	0	8	17	0.87
12	0	0	0	2	2	0.51
13	0	0	0	5	5	0.46
16	0	0	0	4	0	0.36
17	0	0	0	0	4	0.36
20	0	0	0	1	1	0.27
25	1	0	0	8	9	0.25
33	0	0	0	2	2	0.06
54	1	0	0	0	1	0.02

Table 2: Through-deference MOD$_C$(FSAlias) histogram

# Side Effects	Glob	Dyn	Loc	Nv	Total	Percent Below
0	1580	3688	4386	2844	4	100.00
1	2050	460	289	552	2820	99.92
2	37	292	33	125	354	40.40
3	127	94	16	81	158	32.93
4	19	126	14	57	108	29.59
5	35	21	0	35	45	27.31
6	12	3	0	63	58	26.36
7	0	54	0	16	25	25.14
8	47	0	0	19	82	24.61
9	5	0	0	20	93	22.88
10	0	0	0	14	22	20.92
11	0	0	0	5	5	20.45
12	0	0	0	7	2	20.35
13	0	0	0	16	26	20.30
14	0	0	0	12	48	19.76
15	0	0	0	0	2	18.74
17	0	0	0	40	0	18.70
18	0	0	0	0	11	18.70
20	0	0	0	1	1	18.47
25	824	0	0	0	0	18.45
26	0	0	0	1	1	18.45
27	0	0	0	0	41	18.43
77	2	0	0	0	2	17.56
78	0	0	0	6	6	17.52
145	0	0	0	2	0	17.39
146	0	0	0	16	0	17.39
147	0	0	0	32	0	17.39
148	0	0	0	205	0	17.39
149	0	0	0	569	0	17.39
174	0	0	0	0	824	17.39

Table 3: Though-dereference MOD$_C$(FIAlias) histogram

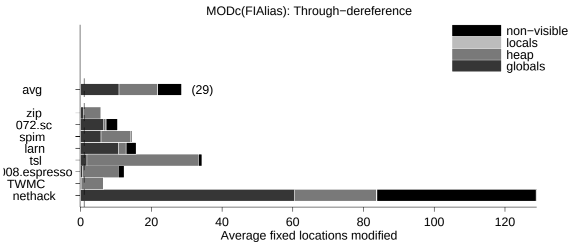


Figure 4: Fixed locations modified by through-dereference assignments

---

the flow- and context-insensitive histogram.

[9] The report of some assignments modifying 0 total locations arises because the front-end analysis does not correctly recognize the dynamic allocation in a declaration of a pointer with an initializer. For example, given `char *a="moose";` the front-end does not recognize `a[1]` as a fixed location. Fortunately, this kind of declaration doesn't occur very often. The front-end does correctly recognize declarations of arrays with initializers.

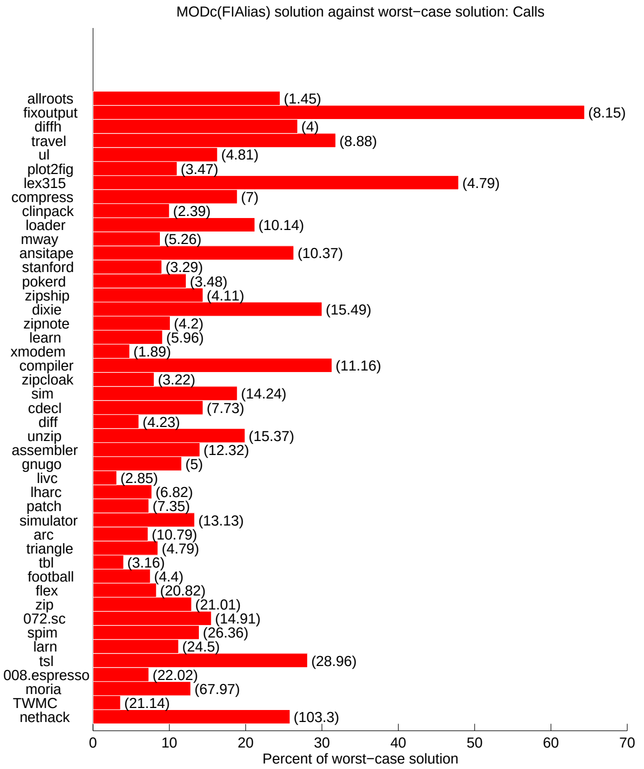


Figure 6: Call MOD$_C$(FIAlias) solution as percentage of worst-case solution.

are reported. Direct assignments do not challenge the data-flow analysis, and the greater proportion of direct assignments to indirect assignments tends to hide the interesting results. The average number over *all* assignment statements of fixed locations reported modified by MOD$_C$(FSAlias) is 1.05. The average for MOD$_C$(FIAlias) is 1.9.[10]

Precision at Call Statements

Figures 5, 6 and 7 present the same information as Figures 2, 3 and 4, respectively, for locations modified by call statements. The conclusions to be drawn are similar. It is not surprising for a call statement to modify several variables. Thus, the notion of modifying close to one location as an indication of precision does not apply to call statement side effects. Nevertheless, as Figure 5(d) shows, the relative gain in precision from flow and context sensitivity is substantial.

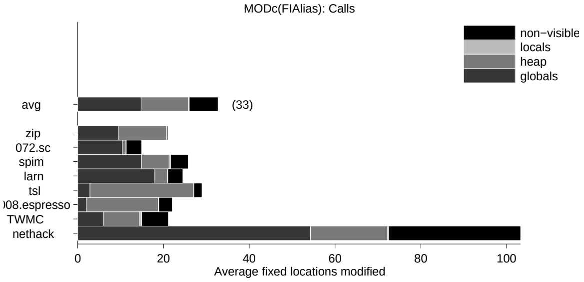


Figure 7: Fixed locations modified by calls

Program	ICFG Nodes	Compile time (s)	$FSAlias$ time (s)	$FIAlias$ time (s)	MOD$_C$ (FS) time (s)	MOD$_C$ (FI) time (s)
allroots	422	2.30	2.46	0.08	0.02	0.02
fixoutput	617	1.72	1.17	0.07	0.03	0.04
diffh	646	1.52	1.74	0.12	0.04	0.04
travel	698	1.90	3.23	0.13	0.04	0.05
ul	1027	1.90	3.10	0.14	0.08	0.06
plot2fig	1077	13.60	3.68	0.19	0.06	0.07
lex315	1297	2.30	3.50	0.14	0.09	0.09
compress	1319	2.50	2.93	0.20	0.08	0.11
clinpack	1429	2.78	7.14	0.25	0.11	0.10
loader	1563	5.40	6.03	0.29	0.12	0.16
mway	1576	2.84	4.31	0.23	0.10	0.12
ansitape	1747	2.92	5.36	0.39	0.11	0.14
stanford	1771	2.38	2.52	0.22	0.09	0.10
pokerd	1895	8.20	24.65	0.29	0.14	0.10
zipship	1955	2.66	4.22	0.28	0.22	0.13
dixie	2341	5.46	7.80	0.31	0.18	0.23
zipnote	2407	6.90	18.72	0.52	0.63	0.16
learn	2626	8.22	6.06	0.37	0.16	0.20
xmodem	2672	6.88	4.75	0.30	0.22	0.19
compiler	3008	6.00	2.91	0.19	0.35	0.35
zipcloak	3033	7.80	20.19	0.64	0.67	0.18
sim	3034	3.12	8.06	0.35	0.31	0.21
cdecl	3196	3.82	21.18	0.31	0.35	0.29
diff	3300	4.60	11.63	0.52	0.32	0.24
unzip	3416	9.94	9.97	0.43	0.26	0.31
assembler	3601	9.12	25.80	0.56	0.50	0.58
gnugo	3651	15.12	3.18	0.32	0.19	0.21
livc	4101	4.30	7.76	0.59	0.27	0.39
lharc	4250	4.56	12.07	0.83	0.32	0.37
patch	4608	7.20	18.53	0.52	0.35	0.45
simulator	5574	11.48	11.42	0.71	0.37	0.56
arc	5856	20.80	14.25	0.93	0.48	0.53
triangle	6119	10.00	8.90	0.66	0.70	0.38
tbl	6162	17.56	27.64	0.56	0.65	0.44
football	7313	10.16	9.72	0.87	0.43	0.50
flex	7376	11.60	111.88	0.76	0.77	0.68
zip	9288	16.44		1.67		0.94
072.sc	13690	16.72		2.06		2.17
spim	16740	22.60		2.12		3.32
larn	21184	36.44		2.43		7.97
tsl	27302	18.98		10.20		13.63
008.espresso	30510	42.64		6.14		5.74
moria	38572	55.60	75.30	7.19	2.88	52.97
TWMC	51627	142.30		9.91		8.87
nethack	58317	79.92		122.75		340.94

Table 4: Comparative Analysis and Compile Times

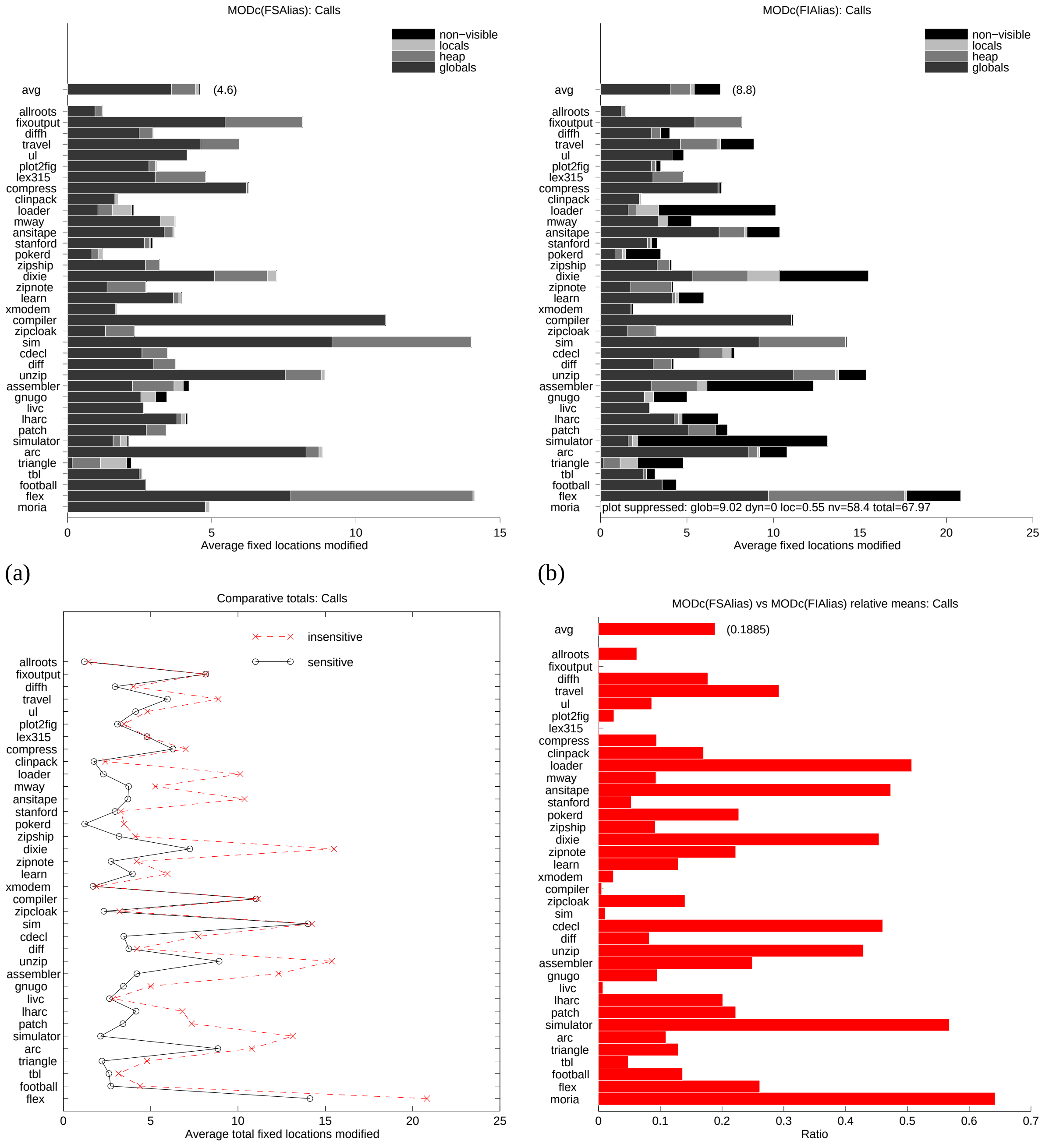
Timing Results

Table 4 shows the analysis times for the MOD$_C$ calculations broken into its two passes, and for a simple compilation using Gnu's gcc compiler version 2.7.2 with no optimizations enabled. These numbers are as reported by the UNIX `time` utility, averaged over 5 executions. The notation MOD$_C$(FS) refers to the phase of the MOD$_C$(FSAlias) algorithm after the alias solution has been computed; similarly for MOD$_C$(FI). Thus, these times in Table 4 *do not* include the alias analysis times, but are simply the time taken to calculate the MOD$_C$ solution given the alias solution. The total analysis time is the sum of the two columns (columns 4 and 6 for MOD$_C$(FSAlias) and columns 5 and 7 for MOD$_C$(FIAlias)).

The first thing to notice about the data in Table 4 is the dramatic difference between $FSAlias$ and $FIAlias$ analysis times. Figure 8 plots the two alias analysis times for the programs with a logarithmic scale on the time axis, and demonstrates at least an order of magnitude difference between the costs of the analyses. The MOD$_C$(FS) and MOD$_C$(FI) are both very fast, and are comparably fast, except in the case of *moria*. Figure 9 shows the MOD$_C$(FS) and MOD$_C$(FI) times with their averages. Notice that seem-

---

[10] Without *moria* the average is 1.3.



(a) (b)

(c) (d)

Figure 5: Fixed locations modified at call statements

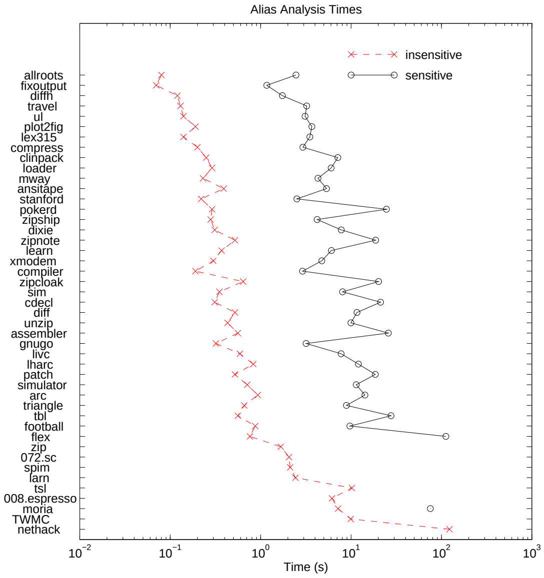


Figure 8: $FSAlias$ analysis time vs $FIAlias$ analysis time

ingly large differences are not large because the scale of the time axis is so small. The only explanation so far for *moria* is that the $FIAlias$ solution is sufficiently imprecise to cause the second pass of the MOD$_C$ algorithm to do significantly more work. Finally, the good news is that in most cases, the MOD$_C$(FSAlias) analysis time is comparable to the compile time. This is an important feature for any analysis destined for compiler optimization. Figure 10 plots the total analysis times for MOD$_C$(FSAlias) ($FSAlias$ + MOD$_C$(FS)) and MOD$_C$(FIAlias) ($FIAlias$ + MOD$_C$(FI)) against the compile times for the data programs. Frequently, the MOD$_C$(FSAlias) analysis is within the same order of magnitude as the compile time, though, of course, there are dramatic exceptions such as *flex* and *nethack*.

RELATED WORK

Banning [1] first accomplished the decomposition of the MOD problem for FORTRAN (and other 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 [4, 5] 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 [2]. All of this work targeted the programming model of Fortran and related languages with no pointer usage.

Choi, Burke, and Carini mention an interprocedural modification-side-effects algorithm for languages with pointers based on their flow-sensitive pointer aliasing technique [3, 12]; it is difficult to compare this work to theirs, because they gave no description of their algorithm and offered no implementation results.

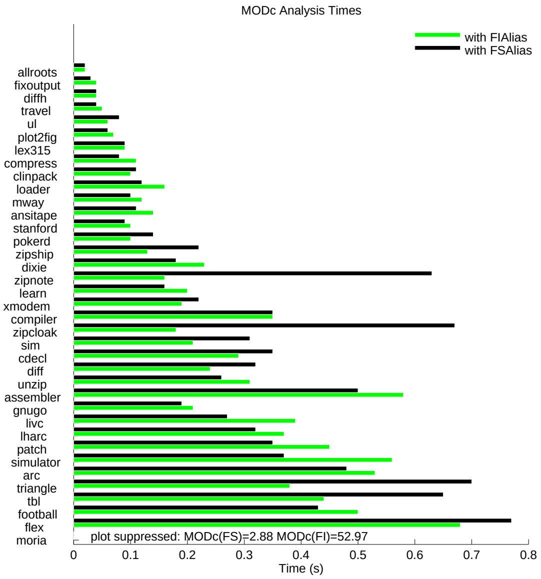


Figure 9: MOD$_C$(FS) analysis time vs MOD$_C$(FI) analysis time

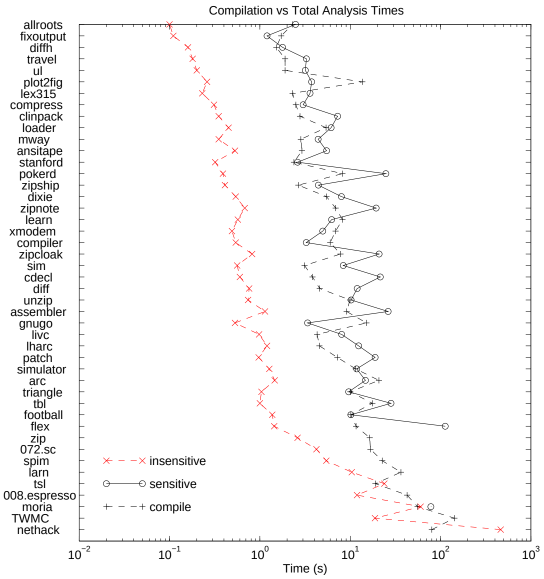


Figure 10: MOD$_C$ analysis times vs compile time

Another approach to side-effect analysis is to perform an interprocedural pointer aliasing algorithm and then identify all variables experiencing side effects at indirect stores through a pointer (i.e., at through-dereference statements) using the aliases found [16, 21]. This is often used as an empirical test of the precision of the alias solution obtained.

Ruf [16] compared the effect of context-sensitivity (or its lack) on a flow-sensitive points-to algorithm. Most of his reported data is with respect to the difference in precision of the points-to solution, with and without context information. This study is difficult to compare with the results presented here because of the use of a lower-level program representation (i.e., VDG vs. ICFG) and differences in the counting of side effects to individual structure fields.

Interprocedural distributive finite subset problems can be solved using a graph reachability technique on an "exploded" call graph of the program [15]. Capture of calling context is not an issue here since the problems being solved are of a form such that reachability in each procedure can be analyzed once for each parameter, regardless of calling context. Several flow- and context-insensitive algorithms for $MOD(P)$, differing by the points-to analysis used, have been profiled in [18]. This study shares the philosophy of the empirical results presented here, in that the effects of pointer aliasing on applications are reported. However, there are no flow- and/or context-sensitive analyses performed and direct comparison with MOD_C(FIAlias) is difficult, since only a flow- and context-insensitive $MOD(P)$ is defined with no per statement side effects, and side effects to structure fields and union members are not distinguished.

OBSERVATIONS and CONCLUSIONS

This is the first empirical comparative study of the effects of both flow and context sensitivity on an important data-flow problem. The obvious conclusion is that flow-/context-sensitive analysis yields significantly more precise solutions at far greater computation cost. Nevertheless, this is a complex and interesting trade-off.

Flow- and context-sensitive analysis

Flow- and context-sensitive data-flow analysis is capable of providing very accurate results for programs of substantial size. As expensive as it is, the cost of sensitive analysis is still not prohibitive for a large subset of the data programs, being on the order of the time to compile the program. Thus, the MOD_C(FSAlias) algorithm achieves scalability up to a certain point.[11] In particular, substantially larger programs that don't use certain program constructs or patterns heavily can be analyzed. The contrast between *moria* and *zip* vividly shows the effects on alias analysis of heavy use of (large) recursive data structures. This level of scalability is rather surprising for a program-point-specific analysis. Further, note that users of software-engineering tools such as data-flow testers or off-line program understanding databases which gather def-use information about a large program in order to query it later, may be willing to accept analysis costs of several times that of the compilation time. Nevertheless, it seems apparent that flow- and context-sensitive analysis is not going to scale to the next order of magnitude without a major innovation. This means that whole-program sensitive analysis of large systems seems unattainable.

Flow- and context-insensitive analysis

Flow- and context-insensitive analysis is a very fast and scalable analysis. Whole program analysis of very large software, such as today's commercial applications, seems feasible. The loss of precision is a strong concern, however. Most applications of the modification-side-effects solution need quite precise results. Nevertheless, it is interesting that the flow- and context-insensitive solutions are very much more precise than the worst-case estimate, meaning that there is still significant gain to be had from using this inexpensive analysis. Software-engineering tools such as smart semantic browsers which trace approximate def-use information or debuggers which use run-time traces augmented by compile-time knowledge are possible consumers of insensitive side-effect information. So, flow-/context-insensitive analysis can be very effective for certain applications, being both accurate and inexpensive.

Comparison of sensitivity

One claim being disputed in the analysis community is that flow- and context-sensitive analysis will obtain much better precision than flow- and context-insensitive analysis on important problems, such as modification side effects.

The empirical results on through-dereference assignments and calls confirm the belief that analysis solution procedure sensitivity provides discernibly increased precision in the solution obtained; for program transformation or testing and validation applications, this accuracy may be required.

Where to now?

This study raises three topics for further exploration. The first is how to incorporate flow and context sensitivity into analysis of very large programs. Zhang *et al.* [21] report on a program decomposition strategy, where the alias relation induces a partitioning of the assignment statements involving pointer variables. This in turn can be used to decompose the program into sections for which analyses of differing precision and cost can be applied; this is especially effective if an expensive analysis can be avoided where much greater accuracy will not be achieved. Initial experiments targeted recursive data structures as subjects for a flow- and context-insensitive alias analysis with appealing results. More experimentation is needed to comprehend the possibilities in this approach, both in terms of choice of analyses to apply to which program sections and also in terms of varying the program decomposition.

The second topic is how to make flow- and context-insensitive analysis more effective without increasing the

---

[11] Previously published results are only up to 4700 lines of code.

cost. An interesting idea stems from the observation that safe analyses produce supersets of the real solution. The intersection of the solutions generated by different, safe analyses for the same problem must also be safe, and may be closer to the real solution. Recently, Shapiro and Horwitz used this idea with several flow- and context-insensitive points-to analysis algorithms [19]. This approach needs more exploratory experimentation.

The final topic is to explore more fully the kind of program construct and programming style that foils data-flow analysis. Perhaps the availability of precise, flow- and context-sensitive data-flow analysis would be sufficient motivation to change programming practice, language design and programmers' habits.

REFERENCES

[1] 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, Jan. 1979.

[2] 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.

[3] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 232–245, Jan. 1993.

[4] 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, Jan. 1985.

[5] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 57–66, June 1988.

[6] N. D. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice Hall, 1982.

[7] 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, Jan. 1982.

[8] 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, Jan. 1991.

[9] 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.

[10] W. Landi, B. G. Ryder, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural side effect analysis with pointer aliasing. Technical Report DCS-TR-336, Department of Computer Science, Rutgers University, Aug. 1997. see http://www.prolangs.rutgers.edu.

[11] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[12] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. Choi, M. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *ACM SIGPLAN Notices*, 28(9):67–70, Sept. 1993.

[13] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.

[14] T. J. Marlowe, B. G. Ryder, and M. Burke. Defining flow sensitivity for data flow problems. Technical Report LCSR-TR-249, Laboratory for Computer Science Research Technical Report, July 1995.

[15] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-second Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 49–61, Jan. 1995.

[16] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995.

[17] E. Ruf. Partitioning data flow analysis using types. In *Conference Record of the Twenty-fourth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 15–26, Jan. 1997.

[18] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *to appear in Proceedings of the Fourth International Symposium on Static Analysis (SAS'97)*, Sept. 1997.

[19] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the Twenty-fourth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 1–14, Jan. 1997.

[20] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[21] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Proceedings of the 4th Annual ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 81–92, Oct. 1996.