RC 21058 (12/15/97)
Computer Science

IBM Research Report

An Empirical Comparison of Interprocedural Pointer Alias Analyses

Michael Hind and Anthony Pioli

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

and

Department of Mathematics and Computer Science
State University of New York at New Paltz
New Paltz, NY 12561

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

An Empirical Comparison of Interprocedural Pointer Alias Analyses*

Michael Hind Anthony Pioli
State University of New York at New Paltz
and
IBM Thomas J. Watson Research Center

Abstract

This paper describes an empirical comparison of three pointer alias analysis algorithms: flow-sensitive, flow-insensitive, and flow-insensitive with precomputed kill information. In addition to contrasting the precision and efficiency of these analyses, it describes implementation techniques and quantifies their analysis-time speed-up. Lastly, it illustrates the object-oriented approach used in the design of the system, which provides a natural example of multiple inheritance.

1 Introduction

In order to effectively analyze programs written in languages that make extensive use of pointers, such as C, C++, or Java (in the form of references), knowledge of pointer behavior is required. Without such knowledge, conservative assumptions must be made, resulting in less precise data flow information, which can adversely affect the effectiveness of analyses that depend on this information.

To increase the precision of program analysis, recent work has developed more ambitious analyses that focus on analyzing the whole program, rather than each procedure in isolation. The main drawback of such approaches is their cost.

A pointer alias analysis is a compile-time analysis that, for each program point, attempts to determine what a pointer can point to. As it is undecidable [20, 29], approximation methods have been developed with varying cost and precision. These algorithms create a spectrum, with the more efficient/less precise algorithms on one end and the less efficient/more precise algorithms on the other. The goal of this paper is to provide data points on this spectrum. Although several researchers have provided empirical results of their techniques, comparisons among algorithms can be difficult because of differing program representations, benchmark suites, and metrics. By holding these factors constant, we can focus more on the efficacy of the algorithms and less on the manner in which the results were obtained.

The contributions of this paper are the following:

- empirical results measuring the precision and efficiency of three pointer alias analysis algorithms,
- insight into the implementation of these algorithms, including empirical evidence of how various implementation enhancements significantly improved analysis time of the flow-sensitive analysis,
- an overview of the object-oriented design of our system, which includes significant use of multiple and virtual inheritance, providing a natural argument for these features.

Section 2 highlights the three algorithms and their implementations. Section 3 describes the empirical study of the three algorithms, analyzes the results, and contrasts them with related results from other researchers. Section 4 overviews some of the performance-improving enhancements made in the implementation and quantifies their analysis-time speed-up. Section 5 illustrates the object-oriented approach used in the design of the system. Section 6 describes other related work. Section 7 draws conclusions and discusses future directions.

2 Analyses and Implementation

2.1 Algorithms

All three analyses implemented in this paper represent the program as a program call (multi-) graph (PCG), in which a node corresponds to a function, and a directed edge represents a call to the target function.[1] Each function body is represented by a control flow graph (CFG). This graph is used to build a simplified sparse evaluation graph (SEG) [9], discussed in Section 4.

Common to all three analyses is the general manner in which they compute their solutions. A nested fixed point

*This material is based upon work supported by the National Science Foundation under grant CCR-9633010, by IBM Research, and by a SUNY at New Paltz Research and Creative Project Award. {hind,pioli}@mcs.newpaltz.edu

[1] Potential calls can occur due to function pointers and virtual methods, where the called function is not known until run-time. For all other calls the edge represents an actual call.

computation is used in which the outer nest corresponds to computing solutions for each function in the PCG. Each such function computation triggers the computation of solutions for each CFG node in the particular function. This general framework is presented as an iterative algorithm in Figure 1 and is further described in [5]. Section 4 reports improvements due to the use of a worklist-based implementation.

All three analyses utilize the *compact representation* [8, 5] to represent alias relations. This representation shares the property of the *points-to* representation [13], in that it captures the "edge" information of alias relations. For example, if variable `a` points to `b`, which in turn points to `c`, the compact representation records only the following alias set: $\{\langle *a, b\rangle, \langle *b, c\rangle\}$, from which it can be inferred that $\langle **\, a, c\rangle$ and $\langle **\, a, *b\rangle$ are also aliases.[2]

All three analyses are *context-insensitive*; they merge information flowing from different calls to the same function, and may suffer from the *unrealizable path problem* [22], i.e., they potentially propagate back to the wrong caller the aliases of the called function. (Sections 3.3 and 4.4 discuss this potential imprecision.)

One manner of classifying interprocedural data flow analyses is whether they consider control flow information. A *flow-sensitive* analysis considers control flow information of a procedure during its analysis of the procedure. A *flow-insensitive* analysis does not consider control flow information during its analysis, and thus can be more efficient, but less precise. (See [26] for a full discussion of these definitions.)

As in [18, 7], all analyses considered here represent the (possibly many) objects allocated at calls to `new` or `malloc` by creating a named object based on the CFG node number of the allocation statement. We refer to these objects as $heap_n$, where n is the CFG node number. These numbers are unique throughout the entire program.

Consider the simple program in Figure 2. The flow-sensitive analysis [8, 5] associates an alias set before (In_n) and after (Out_n) every CFG node, n. For example, $Out_{S1} = \{\langle *p, heap_{S1}\rangle\}$ because $*p$ and $heap_{S1}$ refer to the same storage. At the entry to function `g`, the flow-sensitive analysis will compute $In_{S7} = \{\langle *p, heap_{S4}\rangle\}$, which is the precise solution for this simple example.

The flow-insensitive analysis [4, 5] does not use any intraprocedural control flow information. Instead it conservatively computes what can hold anywhere within a function, i.e., it uses only one alias set per function to represent what may hold at any CFG node in that function. Thus, it must assume that $\langle *p, heap_{S1}\rangle$ and $\langle *p, heap_{S3}\rangle$ can flow into `f`. This results in $In_{S7} = \{\langle *p, heap_{S1}\rangle, \langle *p, heap_{S3}\rangle, \langle *p, heap_{S4}\rangle, \langle *p, heap_{S6}\rangle\}$.

The third analysis, flow-insensitive analysis with kill [4, 5], attempts to improve the precision of the flow-insensitive analysis by *precomputing* kill information for pointers, and then uses this information during the flow-insensitive analysis at call sites.[3] For example, the precomputation will determine that all alias relations involving $*p$ on entry to `f` will be killed before the call to `g` at $S5$. Thus, In_{S7} will contain only the alias relations that could be generated in `f`, i.e., $In_{S7} = \{\langle *p, heap_{S4}\rangle, \langle *p, heap_{S6}\rangle\}$.

As this example illustrates, the flow-sensitive (FS) analysis is at least as precise as the other two, with the flow-insensitive with kill (FIK) analysis being the second most precise. The flow-insensitive (FI) analysis is more efficient than the FIK analysis because it neither precomputes kill information nor uses it during the analysis. One would expect the FS analysis to be the least efficient because it needs to distinguish solutions for every point in the program. Thus, a theoretical spectrum exists in terms of precision and efficiency with the FI analysis on the less precise/more efficient side, the FS analysis on the more precise/less efficient side, and the FIK analysis in the middle.

Not studied in this paper are

- more precise analyses such as a context-sensitive flow-sensitive algorithm [13, 41], or a more precise heap modeling scheme [24, 18, 17, 7, 11, 15, 16, 33];

- more efficient flow-insensitive analyses [38, 35] that only use one alias set for the whole program and limit the number of alias relations by sometimes grouping distinct variables into one named object.

2.2 Implementation

The analyses have been implemented in an experimental program analysis system written in C++. The system utilizes a prototype version of the IBM VisualAge C++ compiler [36, 27] as the front end, which constructs an abstract syntax tree. This tree is transformed into a PCG and a CFG for each function, which serve as input to our algorithms. No CFG is built for library functions. We model a call to a library function based on its semantics. Library calls that cannot affect the value of a pointer are treated as the identity transfer function.

An alias set class that implements the compact representation is used to represent alias relations. All three analyses are implemented using worklists. (Section 4.2 discusses an earlier iterative implementation.) Function pointer analysis is incorporated into the pointer alias analysis as described in [4, 5]. Currently, array elements and field components are not distinguished, and `setjmp/longjmp`[4] and `goto` statements are not supported. The implementation also assumes that pointer values will only exist in pointer variables, and that pointer arithmetic does not result in the pointer going beyond array boundaries.

To model the values passed as `argc` and `argv` to the `main` function, a dummy `main` function was added, which called the benchmark's `main` function, simulating the effects of `argc` and `argv`. This function also initialized the `_iob`

[2] See [25, 5] for a discussion of precision tradeoffs between this representation and a representation that would contain all four pairs.

[3] Kill information is computed in a single flow-sensitive prepass of each CFG. For each call site, c, we compute two sets, the set of pointers that are definitely killed on all paths from entry to c and the set of pointers that are definitely killed on all paths from c to exit. Only the first set is used in our example.

[4] Although one program in our benchmark suite, `anagram`, does syntactically contains a call to `longjmp`, the code is unreachable.

S_1: build the initial PCG
S_2: foreach procedure, p, in the PCG, loop
S_3: initialize interprocedural alias sets of p to $\{\}$
S_4: end loop
S_5: repeat
S_6: foreach procedure, p, in the PCG, loop
S_7: using the interprocedural alias sets (for entry of p and call sites in p), compute the intraprocedural alias sets of p
S_8: using the intraprocedural alias sets of p, update the interprocedural alias sets representing the effect of p on each procedure that calls or is called by p
S_9: end loop
S_{10}: using new function pointer aliases, update the PCG, initializing interprocedural alias sets of new functions to $\{\}$
S_{11}: until the interprocedural alias sets and the PCG converge

Figure 1: High-Level description of General Algorithm [5]

```
    T *p;
    void main(void) {
S1:     p = new T;
S2:     f();
S3:     p = new T;
    }
```

```
    void f(void) {
S4:     p = new T;
S5:     g();
S6:     p = new T;
    }
```

```
    void g(void) {
S7:     ⋮
    }
```

Figure 2: Example program

array, used for standard I/O. The added function is the same as the one added by Ruf [31, 32], which is similar to the one used by Landi et al. [23, 21]. Initializations of global variables are automatically modeled as assignment statements in the dummy `main` function.

3 Results

This section provides empirical evidence of the precision and efficiency for the three algorithms discussed in Section 2.1: flow-sensitive (FS), flow-insensitive (FI), and flow-insensitive with kill (FIK).

3.1 Benchmarks

Our benchmark suite contains fourteen C programs, twelve used by other researchers [23, 13, 31] and two from the SPEC CINT92 [3] and CINT95 [37] benchmarks.[5] Table 1 describes characteristics of the suite. The third column contains the number of lines in the source and header files reported by the Unix utility `wc`. The fourth column reports the number of user-defined functions (nodes in the PCG), which include the dummy `main` function. The next two columns give the number of call sites, distinguished between user and library function calls. The next two columns report cumulative statistics for all CFG nodes and edges. These figures include nodes and edges created by the initialization of globals. The following column computes the ratio of CFG edges to nodes.

The next column reports the percentage of CFG nodes that are considered as pointer-assignment nodes. The analysis treats an assignment as a pointer-assignment if the variable involved in the pointer expression on the left side of the assignment is declared to be a pointer.[6] The last column reports the number of recursive functions in each program.

3.2 Description of Experiment

This section presents precision and efficiency results. Table 2 is similar to Table 3 in [13] and Figure 4 in [31]. For each benchmark and each analysis, we report the analysis time, and the average number of objects the analysis computes a dereferenced pointer can point to. (The precision results for the FIK analysis is exactly the same as the FI analysis for all benchmarks. Thus, to save space we report the FIK analysis times in parenthesis below the FI analysis times.)

The analysis time is reported in seconds. It does not include the time to build the PCG and CFGs, but does include any preprocessing that is specific to the analysis, such as building the SEG from the CFG. The results were collected on an IBM PowerPC 604 machine with 256MB RAM and 240MB paging space, running AIX 4.1.5.

To collect precision information, the system traverses the representation visiting each expression containing one or more dereferences of a pointer and, using the computed alias information, reports how many named objects are aliased to the pointer expression. As in [31], we report the total number of such dereferences for both reads and writes, the average number of objects, and how often 1, 2, 3, or more than 3 objects are returned. The maximum and median number of objects returned are also given. Statements such as "`(*p)++`" and "`*p += increment`" are treated as both a read and a write of `*p`.

We consider a pointer to be dereferenced if the variable is declared as a pointer or an array formal parameter, and one or more of the "`*`", "`->`", or "`[ ]`" operators are used with that variable. Formal parameter arrays are included because their corresponding actual parameter(s) could be a pointer. We do not count the use of the "`[ ]`" operator on arrays that are not formal parameters because the resulting "pointer" (the array name) is constant.

The manner in which the heap is modeled must be considered in evaluating these numbers. For example, a model that uses several names for objects in the heap may seem less precise when compared to a model that uses fewer names [31]. As mentioned above we distinguish heap objects based on their allocation site. Similarly, analyses that represent invisible objects (objects not lexically visible in the current procedure) aliased to a formal parameter (because their address is passed as the corresponding actual parameter) as a single object [22, 13][7] may report fewer objects. Our analyses do not use this technique.

Assuming a correct input program, each pointer dereference should correspond to at least one object at run-time, and thus one serves as a lower bound for the average. Although a precision result close to one demonstrates the analysis is precise (modulo heap and invisible object naming), a larger number could reflect one or more of the following: an imprecise algorithm, a limitation of static analysis, or a pointer dereference where a pointer may point to different addresses over the program's execution.

Table 3 provides alias relation characteristics about dereferenced pointers and is similar to Table 4 in [13]. The first set of columns from "Lo" to "Hp" classifies the type of variable that is dereferenced. For example, in `allroots` 24 of the 40 places where a dereferenced pointer is read the dereferenced variable is a formal parameter. The second half of the table classifies the average number of objects each analysis returns.[8] For example, of the 40 reads through dereferenced pointers in `allroots` both analyses report an average of 1.0 heap object and .60 global objects. The number of heap objects may vary if an alternative heap naming scheme is used.

[5] Some programs had to be syntactically modified to satisfy C++'s stricter type checking semantics. A few program names are different than those reported by Ruf [31]. The SPEC CINT92 program `052.alvinn` was named `backprop` in Todd Austin's benchmark suite [2]. Ruf referred to `ks` as `part`, and `ft` as `span` [32].

[6] This can be conservative in that statements such as "`p->field = ...`" are treated as pointer assignments no matter how the type of `field` is declared. A more accurate categorization would not affect the precision of the analysis, but could improve the efficiency by reducing the number of SEG nodes as discussed in Section 4.1.

[7] This modeling technique can increase the possibility of strong updates [7].

[8] We treat a pointer to a string literal or FILE struct (as returned by `fopen`) as a global.

				Call Sites		CFG			Ptr-Asgn	Recursive
Name	Source	LOC	Funcs	User	Library	Nodes	Edges	$\frac{Edges}{Nodes}$	Nodes	Funcs
allroots	Landi	227	7	19	54	157	174	1.108	2.5%	2
052.alvinn	SPEC92	272	9	8	21	223	272	1.220	9.4%	0
anagram	Austin	650	16	22	60	332	378	1.139	8.7%	1
lex315	Landi	733	17	13	87	527	628	1.192	2.3%	0
ks	Austin	782	14	17	84	513	599	1.168	26.9%	0
09.vor	McCat	1,406	52	174	202	876	926	1.057	27.5%	5
loader	Landi	1,539	30	79	181	687	800	1.164	8.7%	2
ft	Austin	2,156	38	63	118	732	804	1.098	19.7%	0
football	Landi	2,354	58	257	531	2,710	3,124	1.153	1.6%	1
compiler	Landi	2,360	40	349	456	1,723	2,110	1.224	1.2%	14
assembler	Landi	3,446	52	247	490	1,540	1,766	1.147	9.1%	0
yacr2	Austin	3,979	59	158	327	2,030	2,397	1.181	6.7%	5
simulator	Landi	4,639	111	447	673	2,684	3,173	1.182	3.5%	0
099.go	SPEC95	29,637	373	2,054	2,076	16,822	20,758	1.234	.2%	1
Average	—	—	—	—	—	—	—	1.162	9.1%	—

Table 1: Benchmark suite

3.3 Discussion

The results from Table 2 indicate that the flow-insensitive analysis with kill is not beneficial. On our benchmark suite it is never more precise than the flow-insensitive analysis, and on some occasions requires more analysis time than the flow-sensitive analysis. One explanation for the precision result may be that an alias relation created to simulate a reference parameter, where the formal points to the actual, typically is not killed in the called routine, i.e., the formal parameter is not modified, but rather is used to access the passed actual. Thus, programs containing these alias relations will not benefit from the precomputed kill information. We believe the analysis time can be improved using optimizations in the spirit of those described in Section 4.

One surprising result is the overall precision of the flow-insensitive analysis. In half of our benchmarks (those not marked with a "$*$" in Table 2) the flow-insensitive analysis is as precise as the flow-sensitive analysis. This seems to suggest that the added precision obtained by the flow-sensitive analysis in considering control flow within a function is not significant for those benchmarks, at least where pointers are dereferenced. We offer three possible explanations:

1. In general, as the number of control flow edges in a function increases the precision of the flow-sensitive analysis decreases. As this percentage does not affect the flow-insensitive analysis (it does not consider control flow edges), one might expect that programs containing functions with a higher percentage of CFG edges may benefit less from the flow-sensitive analysis. Although further study is needed, this conjecture appears to hold up in our benchmark suite. Four of the six programs over 1,000 LOC that showed a difference in precision had lower than average CFG edge-node ratios. All three of the programs over 1,000 LOC that showed no difference in precision had higher than average CFG edge-node ratios. (Table 1 reports the CFG edge-node ratio.)

2. It appears that a large number of alias relations are created at call sites because of actual/formal parameter bindings. Using the results from Table 3, we see that more than half of the dereferenced pointers are formal parameters.[9] Of the seven programs where the analyses differed in precision, five dereferenced formal parameters more often than the overall average. Thus, the lack of a substantial precision difference between our flow-sensitive and flow-insensitive analyses may be because both algorithms rely on the same (context-insensitive) mapping mechanism at call sites.

3. Pointer variables are often not assigned more than one distinguished object within the same function, therefore the benefit of distinguishing program points is not realized. An exception to this is in the function `InitLists` in the `ks` benchmark, where the same list pointer is used in two list-creating loops. This is reflected in the precision results of Table 2. Further investigation of this issue is warranted.

The precision of the flow-insensitive analysis may come at a slight cost of analysis time. Although the flow-insensitive analysis can be up to 4 times faster than the flow-sensitive analysis (e.g. `compiler`), in one benchmark it was slower (`lex315`).

The precision results from Table 2 for `099.go` merit discussion. An average of 18.35 and 13.64 objects are returned for reads and writes, respectively, with a maximum of 100. This program contains six small list processing functions (using an array-based "cursor" implementation) that accept a pointer to the head of a list as a parameter. One of these functions, `addlist`, is called 404 times, creating 100 different actuals for one of the formals. However, because the scope of the formals is limited to this function (it does not call any other function), these relations are not propagated to any other function. Therefore, these relations do not suffer the effects of the *unrealizable path problem* [22] mentioned in Section 2.1.

[9] The results from Emami et al. [13] also support this claim. In their benchmark suite 95% of dereferenced pointers are formal parameters.

				Num Objects								
Program	Analysis	Time	Type	Avg	Tot	1	2	3	>3	Max	Med	
allroots	FS	0.2	read	1.60	40	16	24	0	0	2	2.0	
			write	1.00	2	2	0	0	0	1	1.0	
227 LOC	FI	0.2	read	1.60	40	16	24	0	0	2	2.0	
7 Functions		(0.3)	write	1.00	2	2	0	0	0	1	1.0	
052.alvinn	FS	0.2	read	1.00	25	25	0	0	0	1	1.0	
			write	1.00	9	9	0	0	0	1	1.0	
272 LOC	FI	0.2	read	1.00	25	25	0	0	0	1	1.0	
9 Functions		(0.3)	write	1.00	9	9	0	0	0	1	1.0	
anagram	FS	1.0	read	1.13	23	20	3	0	0	2	1.0	
			write	1.00	10	10	0	0	0	1	1.0	
650 LOC	FI	0.9	read	1.13	23	20	3	0	0	2	1.0	
16 Functions		(1.5)	write	1.00	10	10	0	0	0	1	1.0	
lex315	FS	0.9	read	1.95	22	1	21	0	0	2	2.0	
			write	1.83	6	1	5	0	0	2	2.0	
733 LOC	FI	1.5	read	1.95	22	1	21	0	0	2	2.0	
17 Functions		(2.5)	write	1.83	6	1	5	0	0	2	2.0	
ks	FS	3.4	read	1.74	72	19	53	0	0	2	2.0	∗
			write	1.62	42	16	26	0	0	2	2.0	∗
782 LOC	FI	1.8	read	1.79	72	15	57	0	0	2	2.0	
14 Functions		(2.9)	write	1.86	42	6	36	0	0	2	2.0	
09.vor	FS	28.0	read	1.68	242	102	115	25	0	3	2.0	∗
			write	1.32	158	119	28	11	0	3	1.0	∗
1,406 LOC	FI	23.9	read	1.75	242	85	132	25	0	3	2.0	
52 Functions		(35.1)	write	1.35	158	113	34	11	0	3	1.0	
loader	FS	9.5	read	1.43	75	48	26	0	1	7	1.0	∗
			write	2.22	37	24	7	1	5	9	1.0	∗
1,539 LOC	FI	3.7	read	1.65	75	46	23	0	6	7	1.0	
30 Functions		(7.1)	write	2.38	37	24	5	1	7	9	1.0	
ft	FS	14.0	read	2.94	79	22	9	0	48	4	4.0	∗
			write	2.03	61	38	3	0	20	4	1.0	∗
2,156 LOC	FI	8.4	read	3.20	79	21	0	0	58	4	4.0	
38 Functions		(12.8)	write	2.18	61	37	0	0	24	4	1.0	
football	FS	5.9	read	1.56	354	256	5	91	2	8	1.0	∗
			write	1.69	16	10	1	5	0	3	1.0	∗
2,354 LOC	FI	5.2	read	1.62	354	249	4	91	10	8	1.0	
58 Functions		(10.2)	write	2.81	16	4	1	5	6	4	3.0	
compiler	FS	14.0	read	1.00	18	18	0	0	0	1	1.0	
			write	0.00	0	0	0	0	0	0	0.0	
2,360 LOC	FI	3.5	read	1.00	18	18	0	0	0	1	1.0	
40 Functions		(5.8)	write	0.00	0	0	0	0	0	0	0.0	
assembler	FS	26.9	read	1.60	211	171	13	0	27	9	1.0	∗
			write	2.16	103	62	16	0	25	9	1.0	∗
3,446 LOC	FI	14.6	read	1.73	211	166	10	0	35	9	1.0	
52 Functions		(28.2)	write	2.37	103	58	14	0	31	9	1.0	
yacr2	FS	33.5	read	1.11	315	281	34	0	0	2	1.0	
			write	1.16	114	97	16	1	0	3	1.0	
3,979 LOC	FI	18.9	read	1.11	315	279	36	0	0	2	1.0	
59 Functions		(36.3)	write	1.25	114	87	26	1	0	3	1.0	
simulator	FS	26.1	read	1.59	129	105	4	12	8	13	1.0	∗
			write	2.05	104	67	11	16	10	13	1.0	∗
4,639 LOC	FI	12.3	read	1.67	129	103	1	14	11	13	1.0	
111 Functions		(20.9)	write	2.09	104	67	9	16	12	13	1.0	
099.go	FS	21.2	read	18.35	57	16	7	3	31	100	5.0	
			write	13.64	47	6	9	9	23	100	3.0	
29,637 LOC	FI	20.1	read	18.35	57	16	7	3	31	100	5.0	
373 Functions		(29.9)	write	13.64	47	6	9	9	23	100	3.0	
Total	FS	—	read	2.13	1,662	1,100	314	131	117	100	—	
		—	write	2.47	709	461	122	43	83	100	—	
	FI	—	read	2.20	1,662	1,060	318	133	151	100	—	
		—	write	2.59	709	424	139	43	103	100	—	

Table 2: Efficiency/precision measurements: numbers in "()" are the FIK analysis times. An entry with a "∗" indicates an increase in precision for the flow-sensitive analysis.

Program	Type	Pointer Type					Analysis	Target Type				
		Lo	Gl	FP	Hp	Tot		Lo	Gl	FP	Hp	Tot
allroots	read	16	0	24	0	40	FS	0.00	0.60	0.00	1.00	1.60
							FI	0.00	0.60	0.00	1.00	1.60
	write	2	0	0	0	2	FS	0.00	0.00	0.00	1.00	1.00
							FI	0.00	0.00	0.00	1.00	1.00
052.alvinn	read	13	0	12	0	25	FS	0.64	0.36	0.00	0.00	1.00
							FI	0.64	0.36	0.00	0.00	1.00
	write	7	0	2	0	9	FS	0.89	0.11	0.00	0.00	1.00
							FI	0.89	0.11	0.00	0.00	1.00
anagram	read	14	2	7	0	23	FS	0.13	0.39	0.00	0.61	1.13
							FI	0.13	0.39	0.00	0.61	1.13
	write	9	0	1	0	10	FS	0.00	0.20	0.00	0.80	1.00
							FI	0.00	0.20	0.00	0.80	1.00
lex315	read	0	3	19	0	22	FS	0.00	0.05	0.00	1.91	1.95
							FI	0.00	0.05	0.00	1.91	1.95
	write	1	4	1	0	6	FS	0.00	0.00	0.00	1.83	1.83
							FI	0.00	0.00	0.00	1.83	1.83
ks	read	59	2	11	0	72	FS	0.00	0.15	0.00	1.58	1.74
							FI	0.00	0.15	0.00	1.64	1.79
	write	25	2	15	0	42	FS	0.05	0.33	0.00	1.24	1.62
							FI	0.05	0.33	0.00	1.48	1.86
09.vor	read	80	31	131	0	242	FS	0.02	0.07	0.00	1.60	1.68
							FI	0.02	0.07	0.00	1.67	1.75
	write	70	37	51	0	158	FS	0.03	0.06	0.00	1.23	1.32
							FI	0.03	0.06	0.00	1.27	1.35
loader	read	22	1	52	0	75	FS	0.29	0.35	0.00	0.79	1.43
							FI	0.29	0.57	0.00	0.79	1.65
	write	4	8	25	0	37	FS	1.60	0.24	0.00	0.38	2.22
							FI	1.60	0.41	0.00	0.38	2.38
ft	read	32	9	38	0	79	FS	0.01	0.72	0.00	2.20	2.94
							FI	0.01	0.85	0.00	2.34	3.20
	write	44	4	13	0	61	FS	0.02	0.39	0.00	1.62	2.03
							FI	0.02	0.46	0.00	1.71	2.18
football	read	12	1	341	0	354	FS	0.05	1.50	0.00	0.00	1.56
							FI	0.05	1.57	0.00	0.00	1.62
	write	4	0	12	0	16	FS	0.06	1.62	0.00	0.00	1.69
							FI	0.06	2.75	0.00	0.00	2.81
compiler	read	5	13	0	0	18	FS	0.00	1.00	0.00	0.00	1.00
							FI	0.00	1.00	0.00	0.00	1.00
	write	0	0	0	0	0	FS	0.00	0.00	0.00	0.00	0.00
							FI	0.00	0.00	0.00	0.00	0.00
assembler	read	70	1	140	0	211	FS	0.62	0.73	0.07	0.18	1.60
							FI	0.62	0.87	0.07	0.18	1.73
	write	16	3	84	0	103	FS	1.47	0.38	0.11	0.20	2.16
							FI	1.47	0.59	0.11	0.20	2.37
yacr2	read	5	247	63	0	315	FS	0.01	0.09	0.00	1.01	1.11
							FI	0.01	0.10	0.00	1.01	1.11
	write	8	77	29	0	114	FS	0.11	0.05	0.00	1.00	1.16
							FI	0.11	0.14	0.00	1.00	1.25
simulator	read	9	38	82	0	129	FS	0.37	0.52	0.35	0.35	1.59
							FI	0.37	0.60	0.35	0.35	1.67
	write	8	36	60	0	104	FS	1.13	0.14	0.35	0.42	2.05
							FI	1.13	0.18	0.35	0.42	2.09
099.go	read	6	0	51	0	57	FS	10.63	7.61	0.11	0.00	18.35
							FI	10.63	7.61	0.11	0.00	18.35
	write	0	0	47	0	47	FS	8.15	5.40	0.09	0.00	13.64
							FI	8.15	5.40	0.09	0.00	13.64
Total	read	20.6%	20.9%	58.4%	0.0%	1662	FS	0.51	0.83	0.04	0.74	2.13
							FI	0.51	0.89	0.04	0.76	2.20
	write	27.9%	24.1%	48.0%	0.0%	709	FS	1.04	0.56	0.07	0.79	2.47
							FI	1.04	0.65	0.07	0.82	2.59

Table 3: Dereferenced pointer characteristics (Lo = local, Gl = global, FP = formal parameter, Hp = heap)

Program	LOC	FS Time	LOC/Sec
09.vor	1,406	28.1	50
loader	1,539	9.5	162
ft	2,156	14.0	154
football	2,354	5.9	399
compiler	2,360	14.0	168
assembler	3,446	26.9	128
yacr2	3,979	33.5	119
simulator	4,639	26.1	178
099.go	29,637	21.2	1,398

Table 4: Flow-sensitive analysis speed for benchmarks over 1,000 LOC

It may seem that a context-sensitive analysis of this function would offer better precision because it would analyze the `addlist` function for each calling context. However, in order to utilize this more precise analysis information, code for 100 separate clones of this function must be generated.

Another conclusion from the results is that analysis time is not only a function of program size; it also depends on the amount of alias relation propagation along the PCG and SEGs. Table 4 illustrates this point for the flow-sensitive analysis. For example, `099.go`, despite being our largest program and having a pointer aliased with 100 objects, is analyzed at the fastest rate (1,398 LOC/second) because these relations are not propagated throughout the program.

As suggested by Shapiro and Horwitz [34] and Diwan [12], a more precise and time-consuming alias analysis may not be as inefficient as it may appear because the time required to obtain increased precision may reduce the time required by subsequent analyses that utilize mod-use information, and thus pointer alias information, as their input. As the previous paragraph suggests, this can also be true about pointer alias analysis itself, which also utilizes pointer alias information during its analysis.

3.4 Comparison with Other Results

Landi et al. [23] report precision results for the computation of the MOD problem using a flow-sensitive pointer alias algorithm with limited context-sensitive information. Among the metrics they report is the number of "thru-deref" assigns, which corresponds to the "write" metrics reported in Table 2. However, since their results include compiler-introduced temporaries in their "thru-deref" count [21], a direct comparison is not possible.

Stocks et al. [39] use the same metric without including temporaries. Using the flow-sensitive context-sensitive analysis of Landi and Ryder [22], the average number of objects ranges from 1.0 to 2.0 on the benchmarks we have in common (`allroots`, `lex315`, `loader`, `football`, `compiler`, `assembler`, `simulator`). On these benchmarks our flow-sensitive context-insensitive analysis ranges from 1.0 to 2.22. Two possible explanations for the slightly less precise results are 1) their algorithm uses some context-sensitivity; 2) the underlying representation is not identical, and thus pointer dereferences may not be counted in the same manner in all cases. For example, in [22] invisible objects aliased to a formal parameter are represented as a single object, and statements such as `cfree(TP)` located in `allroots` are treated as modifying the structure freed, and thus as a pointer dereference [21]. In fact, on the three programs where our analysis reports the same, or close to the same, number of "writes" as "thru-derefs" (`allroots`, `lex315`, `simulator`), our precision is identical to that reported in [39].

The relative precision of the flow-insensitive analysis compared to the flow-sensitive analysis is in contrast to the study of Stocks et al. [39], which compares the flow-sensitive analysis of Landi and Ryder [22] with a flow-insensitive analysis described in [42]. For the seven common benchmarks, our flow-insensitive algorithm ranges from 1.0 to 2.38, compared to 1.0 to approximately 6.3 for the flow-insensitive analysis they studied. We attribute this to the fact that our flow-insensitive analysis does not treat pointer-related assignments as symmetric, as is done in [42].

Emami et al. [13] report precision results for a context and flow-sensitive algorithm. Their results range from 1.0 to 1.77 for all indirect accesses using a heap naming scheme that represents all heap objects with one name. Because we were unable to obtain the benchmarks from their suite, a direct comparison with our results is not possible.

Ruf [31] reports both read and write totals for a flow-sensitive context-insensitive analysis. However, unlike our analysis he counts use of the "[ ]" operator on arrays that are not formal parameters as a dereference [32]. Since such an array will always point to the same place, the average number of objects is improved. For the benchmarks in common,[10] Ruf reports an overall read and write average of 1.35 and 1.43, respectively. To facilitate comparisons, we have also counted in this manner. The results are averages of 1.29 and 1.53, using the common benchmarks. (The results for the complete benchmark suite is contained in Table 7 in the appendix.) We attribute the slight differences to the difference in representations. As Ruf [31] states, "the VDG intermediate representation often coalesces series of structure or array operations into a single memory write." This coalescing can skew results in either direction.

Shapiro and Horwitz [35] present an empirical comparison of three flow-insensitive algorithms. All three algorithms are less precise (but probably more efficient) than the algorithms studied in this paper. Precision is reported in terms of the total number of points-to relations. We agree with [13, 31] that a more meaningful metric is to measure where the points-to information is used as we have done in Table 2.

[10] The common benchmarks are `allroots`, `052.alvinn`, `anagram`, `lex315`, `ks`, `loader`, `compiler`, `assembler`, `yacr2`, and `simulator`. Although [31] reports results for `ft` (under the name `span`), our version of the benchmark is substantially larger than the one Ruf analyzed, and thus is not comparable.

4 Efficiency Improvements

This section describes some of the performance-improving enhancements made in the implementation and quantifies their effects on analysis-time speed-up for the flow-sensitive algorithm. Although novelty is not claimed, the efficacy of each technique is shown.

Table 5 provides program characteristics that contribute to the complexity of pointer alias analysis for each benchmark. The "Back" column is the percentage of CFG nodes that have an incoming back edge — i.e., loop header nodes. The "Merge" column is the percentage of CFG nodes that have more than one predecessor — this includes the nodes counted in the Back column as well as all "end if" and "end switch" nodes. Pointer assignment statements are broken down into three types based on how they affect the Gen and $Kill$ components of their transfer functions. The column marked "1" has both a constant Gen and $Kill$ set. Examples of such statements are "`p = new T`" or "`p = &x`". The column marked "2" has a constant $Kill$ set, but a non-constant Gen set. The statement contains only a pointer variable on the left side, but the right side value is not determined from inspection. Examples of such statements are "`p = q`" or "`p = q->next`." The column marked "3" has neither constant $Kill$ or Gen set. The statement contains a dereference on the left side. Examples of such statements are "`p->next = new T`", "`p->next = q`" or "`p->next = q->next`".[11] These figures do no include the effects of assignment of actual parameters to formal parameters due to call-by-value function calls. The column labeled "Call" is the percentage of call nodes. The remain three columns are discussed in Section 4.1.

4.1 Sharing Alias Sets

As described in Section 2.1, the flow-sensitive analysis computes solutions before and after every node in the CFG. However, since some nodes do not affect pointer aliasing or are not merge points of aliasing information, alias sets can be shared. For example, node #1 in Figure 3 does not affect any pointer value; therefore its Out set will equal its In set. Likewise, a node whose predecessor(s) Out set(s) have the same value will have an In set equal to this value. (For example, all nodes in Figure 3 except #1 and #2.) We use the term *sharing* in a literal way — if the In set is shared with the Out set, they are the *same object* during the analysis. This is done with a shallow copy of the alias set data structures. We precompute these sharing sites in a separate forward pass over the CFG. Each node that has its own set, which we call a *deep* set, is dubbed a "SEG" (*Sparse Evaluation Graph*) node.[12] Such nodes have a list of "SEG" predecessors and "SEG" successors that are used during the analysis. The alias set allocation strategy for a node, n, can be summarized as follows:

[11] Since we do not distinguish structure elements, `p->next` is considered the same as `*p`, thereby ignoring the type of `next`. This can result in an overestimate of the number of type "3" statements.

[12] The method described is a conservative approximation to the SEG of [9] and is similar to [30].

$$In_n = \begin{cases} Out_p & \text{if } \forall p, q \in Pred(n), \\ & p \text{ and } q \text{ share } Out \text{ sets} \\ \text{a deep set} & \text{otherwise} \end{cases}$$

$$Out_n = \begin{cases} In_n & \text{if } n\text{'s transfer function is the identity} \\ & \text{function} \\ \text{a deep set} & \text{otherwise} \end{cases}$$

In addition to saving storage space (on average 75.6% fewer alias sets were allocated), this method saves visits during the actual analysis to nodes that can not affect pointer aliasing. The last three columns of Table 5 report the number of alias sets before and after this optimization was applied as well as the percentage reduction.

Table 6 shows the effects of the efficiency improving ideas on the flow-sensitive analysis. The headings for each column are read vertically — for example, the second column shows the analysis time in seconds with no sharing of alias sets as well as no other enhancements described in this section. The third column reports the analysis time and speed-up using this sharing technique.

The effectiveness of this technique is mostly related to the percentage of CFG nodes that affect a pointer and percentage of call nodes — the higher the percentages the lower the potential benefit.[13] For all our benchmarks, these averages were 9.1% and 22% respectively. (`09.vor` had high averages in these categories and has the lowest effectiveness for this technique while `yacr2` had lower than average percentages and resulted in a higher speed-up.)

Over all benchmarks, this technique results in a significant speed-up, 2.78 on average. For our largest benchmark, `099.go`, the analysis did not run (due to insufficient memory) until we applied this efficiency optimization.

4.2 Worklists

The initial implementation of the analyses used an iterative algorithm for simplicity. After correctness was verified, a worklist-based implementation was used to improve efficiency. Two types of worklists were used: SEG node worklists and function worklists. Each function has a worklist of SEG nodes. The PCG has two worklists of functions: "current" and "next". (The motivation for using two worklists will be described in Section 4.3.)

We use nested "while not empty" structures with the worklists — the outer one visiting functions and the inner visiting SEG nodes. The worklist of functions initially contains all functions. On the first visit to a function, we initialize the function's SEG node worklist with all SEG nodes in that function. If a SEG node's Out set changes, all its SEG successors are placed on its function's SEG node worklist. If a call to `ForwardBind()` (the function that propagates aliases from call nodes to the called function) changes the entry set of the

[13] The percentage of back edges and merge nodes plays a smaller role.

Benchmark	CFG Nodes								# Alias Sets		
	Total #	Pct Back	Pct Merge	# Ptr Assign Type 1	2	3	Total Pct	Pct Call	No Seg	SEG	Pct Saved
allroots	157	4.5%	10.8%	3	0	1	2.5%	34%	328	87	73.5%
052.alvinn	223	13.0%	13.0%	7	14	0	9.4%	9%	464	89	80.8%
anagram	332	4.8%	13.9%	6	15	8	8.7%	18%	696	159	77.2%
lex315	527	2.5%	15.6%	5	2	5	2.3%	29%	1,088	259	76.2%
ks	513	6.8%	12.7%	19	88	31	26.9%	16%	1,054	318	69.8%
09.vor	876	1.0%	9.9%	13	79	149	27.5%	23%	1,856	633	65.9%
loader	687	3.6%	14.7%	11	20	29	8.7%	26%	1,434	392	72.7%
ft	732	3.3%	10.5%	20	68	56	19.7%	16%	1,540	381	75.3%
football	2,710	3.7%	12.9%	4	30	9	1.6%	20%	5,536	960	82.7%
compiler	1,723	1.5%	17.6%	6	14	0	1.2%	26%	3,526	701	80.1%
assembler	1,540	2.0%	13.8%	22	41	77	9.1%	32%	3,184	933	70.7%
yacr2	2,030	6.1%	14.9%	28	10	99	6.7%	16%	4,178	831	80.1%
simulator	2,684	1.7%	17.3%	19	16	58	3.5%	25%	5,590	1,227	78.1%
099.go	16,822	4.1%	21.4%	5	1	23	0.2%	12%	—	4,968	—
Averages	—	4.2%	14.2%	—	—	—	9.1%	22%	—	—	75.6%

Table 5: Benchmark characteristics. The "Back" column is % of CFG nodes that have an incoming back edge — i.e., loop header nodes. The "Merge" column is % of CFG nodes that have more than one predecessor — this includes all the nodes counted in the Back column and adds all "end if" and "end switch" nodes.

Benchmark Name	No Sharing	←− Sharing −→				Overall Speed-up
	←− Iterating −→		←− Worklists −→			
			Unsorted	Sorted		
	←− No Forward Bind Filter −→				Filter	
allroots (227 LOC)	3.08	1.26 (2.44)	0.44 (2.86)	0.47 (0.94)	0.23 (2.04)	13.40
052.alvinn (272 LOC)	2.20	0.83 (2.65)	0.21 (3.95)	0.21 (1.00)	0.21 (1.00)	10.48
anagram (650 LOC)	11.45	3.85 (2.97)	1.24 (3.10)	1.11 (1.12)	1.01 (1.10)	11.34
lex315 (733 LOC)	10.28	4.42 (2.33)	1.39 (3.18)	1.37 (1.01)	0.86 (1.59)	11.95
ks (782 LOC)	49.12	21.91 (2.24)	7.69 (2.85)	6.15 (1.25)	3.36 (1.83)	14.62
09.vor (1,406 LOC)	575.21	304.15 (1.89)	98.94 (3.07)	88.66 (1.12)	28.01 (3.17)	20.54
loader (1,539 LOC)	427.88	149.96 (2.85)	65.94 (2.27)	55.50 (1.19)	9.54 (5.82)	44.86
ft (2,156 LOC)	206.32	95.16 (2.17)	35.51 (2.68)	28.85 (1.23)	14.04 (2.05)	14.70
football (2,354 LOC)	655.85	150.57 (4.36)	63.11 (2.39)	55.99 (1.13)	5.94 (9.42)	110.41
compiler (2,360 LOC)	139.52	47.64 (2.93)	17.15 (2.78)	16.09 (1.07)	14.00 (1.15)	9.96
assembler (3,446 LOC)	1601.92	649.98 (2.46)	317.94 (2.04)	259.93 (1.22)	26.94 (9.65)	59.46
yacr2 (3,979 LOC)	1035.09	278.19 (3.72)	83.13 (3.35)	84.68 (0.98)	33.54 (2.52)	30.86
simulator (4,639 LOC)	1200.11	391.38 (3.07)	211.37 (1.85)	166.22 (1.27)	26.13 (6.36)	45.93
099.go (29,637 LOC)	— —	595.91 —	181.43 (3.28)	168.50 (1.08)	21.24 (7.93)	28.06
Average Speed-up	—	2.78	2.83	1.12	3.97	30.47

Table 6: Flow-sensitive analysis run-time in seconds. Numbers in parentheses are the speed-up from the previous version to the left.

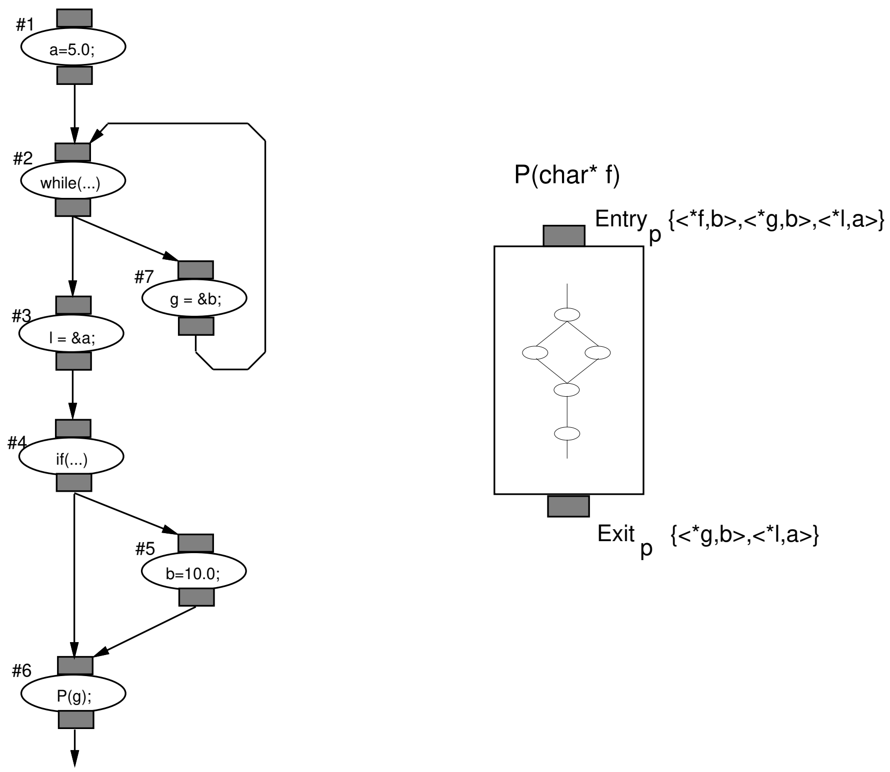

Figure 3: Example CFG (left) and function(right). Note that `g` is a global, `a,b,l` are locals of the CFG, and `f` is a formal. Possible topological numbers (ignoring the back edge $7 \rightarrow 2$) appear next to each node.

called function, the called function is placed on the "next" function worklist. If the exit set of a function changes, all calling functions are placed on the "next" function worklist and the calling call node(s) are placed on their respective function's SEG node worklist. The analysis runs until all worklists are empty.

Column four of Table 6 shows the improvement of the worklist-based implementation over the iterating version, both of which use shared alias sets. The result was an average speed-up of 2.83 over the iterating version.

4.3 Sorted Worklists

Although a worklist implementation improved analysis time, we discovered that it too could be improved. Using an iterating analysis of a forward problem, it is natural to process the nodes in topological order. The next enhancement was to use priority queues (based on a topological order) for the SEG and function worklists.

Consider, however, the case where we have a loop body that generates aliasing information. It would be best to process the loop body before moving on to the loop exit. Topological order alone does not give this property — we may process the loop exit before the loop body. (This would occur in Figure 3 with the topological order as shown.) However, during the construction of the CFG, nodes for loop bodies are created before those nodes after the loop body. (Thus, node #7 would be created before node #3.) Since nodes are assigned numbers as they are created (which is performed in a topological order), using the nodes creation number as a priority ensures that loop bodies are processed first.

A result of using a single priority-based worklist of functions was that calling functions were visited before called functions. However, unlike SEG nodes, aliases can be propagated in both directions along a PCG edge. Thus, an optimal order of function visits is not apparent. In our benchmark suite, using a single priority-based worklist for functions provided only a marginal improvement.

To increase efficiency, we use two function worklists — "current" and "next". While visiting functions on the "current" worklist, we place functions on the "next" worklist. This has the effect that once a set of functions is on the worklist, the visiting order is fixed in a topological order. When the "current" worklist is empty, we swap the "current" and "next" worklists and continue the analysis.

The fifth column of Table 6 reports the analysis time using sorted worklists for both SEG nodes and functions. This enhancement resulted in an average of 1.12 speed-up over the previous version, which used non-priority based worklists.

4.4 ForwardBind Filtering

`ForwardBind()` propagates alias relations from call nodes to called functions. If needed, it creates alias relations from the formals to the actuals in the called function's entry set and then unions in the call node's In set with the called function's entry set. When the called function is analyzed, these aliases can be propagated through each node in its SEG. Although correct, this is inefficient. Consider those alias relations that cannot be changed or used in the called function (for example, the relation $\langle *l, a\rangle$ in Figure 3), but are still propagated through the called function until they reach the exit set of the function. These relations are then propagated back to the call node's Out set. In effect, the called function acts as an identity transfer function for those relations that are not relevant in the called function.

Our enhancement is to not propagate from the call node those alias relations that cannot be reached from a called function. To compute the set of alias relations that are not reachable in the called function, F, we first call `ForwardBind()`, which propagates alias relations into the entry set of F, $Entry_F$, as described above. We then view all alias relations in $Entry_F$ as a directed graph and remove from this graph all vertices (and associated edges) that cannot be reached from a global or a formal of F. These edges (i.e., alias relations) are simply unioned into the call node's Out set directly. This can help limit the propagation effects of the unrealizable path problem.

The last column of Table 6 reports the effectiveness of this optimization. Its improvement provided the most dramatic speed-up, an average of 3.97. Some programs, in particular `football` and `assembler`, had a much higher than average speed-up resulting from this filtering. These two programs both shared some common characteristics: they have a single function that has both an unusually high amount of pointer affecting statements and a very high number of called functions.

The cumulative effect of all five optimizations was an average speed-up of 30.47. This illustrates the benefits that can be obtained by limiting the propagation of extraneous alias relations and the number of visits to functions and nodes.

5 System Overview and Design

Our program representation, called Dakota, was designed to enable various intra- and interprocedural data flow analyses with extensibility being an important goal. In designing the classes used for the three analyses a recurring inheritance pattern emerged. Figure 4 illustrates this pattern in a C++ class hierarchy graph.

Class `Node`, an abstract base class, represents a CFG node that includes a list of predecessor and successor nodes. Class `BinaryAssignNode` virtually inherits from `Node` and adds assignment statement information, such as expression trees for the left and right side of the assignment. Similar classes exist for other types of CFG nodes. These classes represent a generic CFG and are not specific to any data flow analysis.

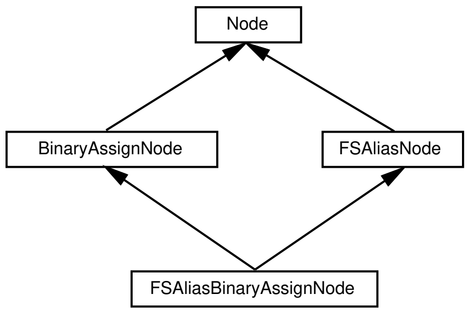

Figure 4: Sample inheritance pattern

To implement the flow-sensitive analysis, we first define a class `FSAliasNode`, which virtually inherits from `Node` and adds the In set and Out set data members needed for the flow-sensitive analysis. In addition, it implements a default *meet* [1] method to compute the In set for all flow-sensitive nodes.[14] Class `FSAliasBinaryAssignNode` inherits the In and Out sets from `FSAliasNode` and the assignment statement information from `BinaryAssignNode`. It computes the Out set by applying the transfer function to the In set using the assignment statement information.

A similar inheritance pattern is used in the implementation of the flow-sensitive analysis for other types of nodes, expressions, and functions, as well as in the flow-insensitive analyses. This inheritance pattern can be generalized to more than one simultaneous data flow analysis [28].

Other features of the system include

- Dakota makes extensive use of the *Abstract Factory* and *Factory Method* patterns [14] to give analysis clients a convenient mechanism for constructing the appropriate objects.
- Analysis and front-end independence. The representation provides basic methods that are needed by most data flow analyses, e.g., CFG nodes have lists of successors and predecessors.
- Persistent storage. The entire program representation can be saved to and restored from ASCII files. This also gives Dakota platform independence.

6 Other Related Work

This section describes related work that has not been mentioned in Section 3.4 and other sections of the paper.

[14] As the meet method needs the list of CFG predecessors, it is natural to inherit this information from `Node`.

Ruf [31] presents an empirical study of two algorithms: a flow-sensitive algorithm similar to the one we have implemented, and a context-sensitive version of the same algorithm. His results showed that the context-sensitive algorithm did not improve precision for pointers where they are dereferenced, but cautioned that this may be a characteristic of the benchmark suite analyzed. Wilson and Lam [41] present an algorithm for performing context-sensitive analysis, but attempt to avoid redundant analyses of functions for similar calling contexts. Horwitz [19] defines precise flow-insensitive alias analysis and proves that, even in the absence of dynamic memory allocation, computing it is NP-hard.

Choi et al. [8] describes the flow-sensitive algorithm implemented in this paper, and provides a high-level description of the two flow-insensitive algorithms. A more detailed description of the flow-insensitive algorithms can be found in [4]. These algorithms and other pointer aliasing techniques are further described in [5].

The Vortex [10, 6] and SUIF [40] compilers provide infrastructures for interprocedural analyses. For efficiency reasons, they do not use the multiple inheritance-based scheme outlined in Section 5, but provide the possibility of augmenting the equivalent of a CFG node with a list of analysis-specific methods to apply to that node.

7 Conclusions and Future Work

This work has described an empirical study of three pointer alias analysis algorithms. After accounting for differences in program representation and metrics, we have found that the precision of most flow-sensitive analyses are roughly equivalent. We have also shown that a flow-sensitive analysis can be efficiently implemented to handle programs on the order of 30,000 LOCs. We have empirically demonstrated how various implementation strategies result in significant analysis-time speed-up. Finally, we have described how the design of our system offers an example of multiple inheritance.

In the future we would like to:

- Study a larger collection of benchmarks, including those written in C++,
- Implement a more efficient/less precise flow-insensitive algorithm and see how it compares in efficiency and precision to our flow-insensitive algorithm,
- Investigate a more efficient function visiting order,
- Extend the system to distinguish field components of the same object.

8 Acknowledgements

We thank Lee Nackman for his support of this work. We thank Todd Austin, Bill Landi, and Rakesh Ghiya for making their benchmarks available.

Bill Landi, Erik Ruf, and Bob Wilson provided useful details concerning their implementations. Michael Burke, Paul Carini, and Jong-Deok Choi provided useful discussions regarding the algorithms described in [8].

Harini Srinivasan designed and implemented the initial Dakota control flow graph builder, an important early system component. David Bacon helped with the initial design of the program call graph representation. NPIC group members Robert Culley, Lynne Delesky, Lap Chung Lam, Giampaolo Lauria, Devasena Padmanaban, Mark Nicosia, Keith Sanders, Truong Vu, and Ming Wu assisted with the implementation and testing of the system.

Michael Burke, Jong-Deok Choi, John Field, G. Ramalingam, Harini Srinivasan, and Laureen Treacy provided useful comments on earlier drafts of this paper.

References

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

[2] Todd Austin. Pointer-intensive benchmark suite, version 1.1. http://www.cs.wisc.edu/~austin/ptr-dist.html, 1995.

[3] Subra Balan and Walter Bays. Spec announces new benchmark suites cint92 and cfp92. Technical report, Systems Performance Evaluation Cooperative, March 1992. *SPEC Newsletter* 4(1).

[4] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science, 892*, pages 234–250. Springer-Verlag, 1995. Proceedings from the *7th International Workshop on Languages and Compilers for Parallel Computing*. Extended version published as Research Report RC 19546, IBM T. J. Watson Research Center, September, 1994.

[5] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Interprocedural pointer alias analysis. Research Report, IBM T.J. Watson Research Center, November 1997.

[6] Craig Chambers, Jeffrey Dean, and David Grove. Frameworks for intra- and interprocedural dataflow analysis. Technical Report UW-CSE-96-11-02, Department of Computer Science and Engineering, University of Washington, November 1996.

[7] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990. *SIGPLAN Notices 25(6).*

[8] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 232–245, January 1993.

[9] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *18th Annual ACM Symposium on the Principles of Programming Languages*, pages 55–66, January 1991.

[10] Jeffrey Dean, Greg DeFouw, David Grove, Vass Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1996.

[11] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994. *SIGPLAN Notices, 29(6).*

[12] Amer Diwan. Personal communication, August 1997.

[13] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994. *SIGPLAN Notices, 29(6).*

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[15] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[16] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag or a cyclic graph? A shape analysis for heap-directed pointers in C. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, January 1996.

[17] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[18] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 28–40, June 1989. *SIGPLAN Notices* 24(6).

[19] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-Hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997.

[20] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[21] William Landi. Personal communication, October 1997.

[22] William Landi and Barbara Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992. *SIGPLAN Notices* 27(6).

[23] William Landi, Barbara Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993. *SIGPLAN Notices* 28(6).

[24] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, 1988.

[25] Thomas Marlowe, William Landi, Barbara Ryder, Jong-Deok Choi, Michael Burke, and Paul Carini. Pointer-induced aliasing: A clarification. *SIGPLAN Notices*, 28(9):67–70, September 1993.

[26] Thomas J. Marlowe, Barbara G. Ryder, and Michael G. Burke. Defining flow sensitivity in data flow problems. Technical Report RC 20138 (89049), IBM Thomas J. Watson Research Center, July 1995.

[27] Lee R. Nackman. Codestore and incremental C++. *Dr. Dobbs Journal*, pages 92–95, December 1997.

[28] Anthony Pioli. Conditional pointer aliasing and constant propagation. Master's thesis, SUNY at New Paltz, 1998. In preparation.

[29] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(6):1467–1471, November 1994.

[30] G. Ramalingam. On sparse evaluation representations. In Pascal Van Hentenryck, editor, *Lecture Notes in Computer Science, 1302*, pages 1–15. Springer Verlag, 1997. Proceedings from the *4th International Static Analysis Symposium*.

[31] Erik Ruf. Context-insensitive alias analysis reconsidered. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, 1995. *SIGPLAN Notices, 30(6).*

[32] Erik Ruf. Personal communication, October 1997.

[33] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, January 1996.

[34] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In Pascal Van Hentenryck, editor, *Lecture Notes in Computer Science, 1302*, pages 16–34. Springer-Verlag, 1997. Proceedings from the *4th International Static Analysis Symposium*.

[35] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive point-to analysis. In *24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, 1997.

[36] D. Soroker, M. Karasick, J. Barton, and D. Streeter. Extension mechanisms in Montana. In *8th IEEE Israeli Conference on Software and Systems*, June 1997.

[37] SPEC. CPU95, Version 1.0. Standard Performance Evaluation Corporation, http://www.specbench/org, August 1995.

[38] Bjarne Steensgaard. Points-to analysis in almost linear time. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, 1996.

[39] Philip A. Stocks, Barbara G. Ryder, William A. Landi, and Sean Zhang. Comparing flow and context sensitivity on the modifications-side-effects problem. Technical Report DCS-TR-335, Department of Computer Science, Rutgers University, August 1997.

[40] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[41] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, 1995. *SIGPLAN Notices, 30(6)*.

[42] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *Symposium on the Foundations of Software Engineering*, pages 81–92, 1996.

A Appendix

Table 7 reports results for all three analyses counting all uses of "$*$", "$->$", "[ ]" as dereferences as is done in [31]. Once again, the precision numbers for the flow-insensitive analysis with kill are identical to the version without kill and are omitted.

Program	Analysis	Type	Avg	Num Objects							
				Tot	1	2	3	>3	Max	Med	
allroots	FS	read	1.56	43	19	24	0	0	2	2.0	
		write	1.00	4	4	0	0	0	1	1.0	
227 LOC	FI	read	1.56	43	19	24	0	0	2	2.0	
7 Functions		write	1.00	4	4	0	0	0	1	1.0	
052.alvinn	FS	read	1.00	44	44	0	0	0	1	1.0	
		write	1.00	22	22	0	0	0	1	1.0	
272 LOC	FI	read	1.00	44	44	0	0	0	1	1.0	
9 Functions		write	1.00	22	22	0	0	0	1	1.0	
anagram	FS	read	1.06	49	46	3	0	0	2	1.0	
		write	1.00	28	28	0	0	0	1	1.0	
650 LOC	FI	read	1.06	49	46	3	0	0	2	1.0	
16 Functions		write	1.00	28	28	0	0	0	1	1.0	
lex315	FS	read	1.81	26	5	21	0	0	2	2.0	
		write	1.50	10	5	5	0	0	2	1.5	
733 LOC	FI	read	1.81	26	5	21	0	0	2	2.0	
17 Functions		write	1.50	10	5	5	0	0	2	1.5	
ks	FS	read	1.47	113	60	53	0	0	2	1.0	*
		write	1.39	66	40	26	0	0	2	1.0	*
782 LOC	FI	read	1.55	113	51	62	0	0	2	2.0	
14 Functions		write	1.56	66	29	37	0	0	2	2.0	
09.vor	FS	read	1.68	242	102	115	25	0	3	2.0	*
		write	1.31	160	121	28	11	0	3	1.0	*
1,406 LOC	FI	read	1.75	242	85	132	25	0	3	2.0	
52 Functions		write	1.35	160	115	34	11	0	3	1.0	
loader	FS	read	1.41	78	51	26	0	1	7	1.0	*
		write	1.96	47	34	7	1	5	9	1.0	*
1,539 LOC	FI	read	1.63	78	49	23	0	6	7	1.0	
30 Functions		write	2.09	47	34	5	1	7	9	1.0	
ft	FS	read	3.09	92	22	9	0	61	4	4.0	*
		write	2.16	70	41	3	0	26	4	1.0	*
2,156 LOC	FI	read	3.32	92	21	0	0	71	4	4.0	
38 Functions		write	2.33	70	39	0	0	31	4	1.0	
football	FS	read	1.12	1629	1531	5	91	2	8	1.0	*
		write	1.05	201	195	1	5	0	3	1.0	*
2,354 LOC	FI	read	1.14	1629	1524	4	91	10	8	1.0	
58 Functions		write	1.14	201	189	1	5	6	4	1.0	
compiler	FS	read	1.15	97	82	15	0	0	2	1.0	
		write	1.18	51	42	9	0	0	2	1.0	
2,360 LOC	FI	read	1.15	97	82	15	0	0	2	1.0	
40 Functions		write	1.18	51	42	9	0	0	2	1.0	
assembler	FS	read	1.59	213	173	13	0	27	9	1.0	*
		write	1.98	121	80	16	0	25	9	1.0	*
3,446 LOC	FI	read	1.73	213	168	10	0	35	9	1.0	
52 Functions		write	2.17	121	76	14	0	31	9	1.0	
yacr2	FS	read	1.11	315	281	34	0	0	2	1.0	*
		write	1.16	116	99	16	1	0	3	1.0	*
3,979 LOC	FI	read	1.11	315	279	36	0	0	2	1.0	
59 Functions		write	1.24	116	89	26	1	0	3	1.0	
simulator	FS	read	1.24	460	425	4	12	19	13	1.0	*
		write	1.65	229	179	11	16	23	13	1.0	*
4,639 LOC	FI	read	1.28	460	423	1	14	22	13	1.0	
111 Functions		write	1.72	229	179	9	16	25	13	1.0	
go	FS	read	1.13	7,896	7,855	7	3	31	100	1.0	
		write	1.48	1,236	1,195	9	9	23	100	1.0	
29,637 LOC	FI	read	1.13	7,896	7,855	7	3	31	100	1.0	
373 Functions		write	1.48	1,236	1,195	9	9	23	100	1.0	
Total	FS	read	1.17	11,297	10,696	329	131	141	100	—	
		write	1.47	2,361	2,085	131	43	102	100	—	
	FI	read	1.19	11,297	10,651	338	133	175	100	—	
		write	1.51	2,361	2,046	149	43	123	100	—	

Table 7: Results counting all uses of "$*$", "$->$", "[ ]". An entry with a "$*$" indicates an increase in precision for the flow-sensitive analysis.