RC 21251 (94735) 6/24/98
Computer Science

IBM Research Report

Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses (Extended Version)

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

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

Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses* (Extended Version)†

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 four context-insensitive pointer alias analysis algorithms that use varying degrees of flow-sensitivity: a flow-insensitive algorithm that tracks variables whose addresses were taken and stored; a flow-insensitive algorithm that computes a solution for each function; a variant of this algorithm that uses precomputed kill information; and a flow-sensitive algorithm. In addition to contrasting the precision and efficiency of these analyses, we describe implementation techniques and quantify 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

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 about pointer values must be made, resulting in less precise data flow information, which can adversely affect the effectiveness of analyses and tools 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 such an analysis is, in general, undecidable [29, 38], approximation methods have been developed. These algorithms provide trade-offs between the efficiency of the analysis and the precision of the computed solution. The goal of this work is to quantify how the use of flow-sensitivity affects precision and efficiency.

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 that measure the precision and efficiency of four pointer alias analysis algorithms with varying degrees of flow-sensitivity;
- 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.

In addition to the use of flow-sensitivity, other factors that affect the cost/precision trade-offs of pointer alias analyses include the use of context-sensitivity and the manner in which aggregates (arrays and structs) and the heap are modeled. Our experiments hold these factors constant so that the results only reflect the usage of flow-sensitivity.

Section 2 highlights the four algorithms and their implementations. Section 3 describes the empirical study of the four 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 6 describes other related work. Section 7 states conclusions.

2 Analyses and Implementation

One manner of classifying interprocedural data flow analyses is whether they consider control flow information during the analysis. A *flow-sensitive* analysis considers control flow information of a procedure during its analysis of the procedure.

*This work was supported in part by the National Science Foundation under grant CCR-9633010, by IBM Research, and by SUNY at New Paltz Research and Creative Project Awards. EMail: {hind,pioli}@mcs.newpaltz.edu, URL: http://www.mcs.newpaltz.edu/~hind, http://www.ulster.net/~pioli

†This paper is an extension of [25].

A *flow-insensitive* analysis does not consider control flow information during its analysis, and thus can be more efficient, but less precise. (See [35] for a full discussion of these definitions.)

The algorithms we consider, listed in order of increasing precision, are

AT: a flow-insensitive algorithm that computes one solution set for the entire program that contains all named objects whose address has been taken and stored,

FI: a flow-insensitive algorithm [5, 6] that computes a solution set for every function,

FIK: a flow-insensitive algorithm [5, 6] that computes a solution set for every function, but attempts to improve precision by using precomputed (flow-sensitive) kill information,

FS: a flow-sensitive algorithm [10, 6] that computes a solution set for every program point.

The following sections provide further information about these analyses and their implementation.

2.1 Algorithms

The program is represented 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) [11], discussed in Section 4.

The address-taken analysis (AT) computes its solution by making a single pass over all functions in the program,[2] adding to a global set all variables whose addresses have been assigned to another variable. These include actual parameters whose addresses are stored in the corresponding formal. Examples are statements such as "`p = &a;`", "`q = new ...;`", and "`foo(&a);`", but not simple expression statements such as "`&a;`" because the address was not stored. AT is efficient because it is linear in the size of the program and uses a single solution set, but it can be very imprecise. It is provided as a base case for comparison to the other three algorithms presented in this paper.

The general manner in which the other three analyses compute their solutions is the same. A nested fixed point 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 a local solution for all program points that are distinguished in the particular analysis. For the flow-sensitive (FS) analysis, the local solution corresponds to each CFG node in the function. For the other two flow-insensitive analyses (FI, FIK), the local solution corresponds to one set that conservatively represents what can hold anywhere in the function. This general framework is presented as an iterative algorithm in Fig. 1 and is further described in [6]. An extension to handle virtual methods is described in [7]. Section 4 reports improvements due to the use of a worklist-based implementation.

The FS, FI, and FIK analyses utilize the *compact representation* [10, 6] to represent alias relations. This representation shares the property of the *points-to* representation [17], 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.[3]

All analyses are *context-insensitive*; they merge information flowing from different calls to the same function, and may suffer from the *unrealizable path problem* [31], 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.) Context-sensitive analyses [17, 53] do not suffer from this problem, but may increase time/space costs.

As in [26, 9], 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. These objects are referred to as $heap_n$, where n is the CFG node number of the allocation statement. These names are unique throughout the entire program. More precise heap modeling schemes [33, 26, 23, 9, 10, 14, 19, 20, 43, 44] can improve precision, but may also increase time/space costs. Quantifying the effects of using context-sensitivity and various heap models is beyond the scope of this work.

Consider the simple program in Fig. 2. The AT analysis computes only one set of objects, which it assumes all pointers may point to. This set will contain five objects $\{heap_{S1}$, $heap_{S3}$, $heap_{S4}$, $heap_{S6}$, and $heap_{S7}\}$, all of which will appear to be referenced at $S8$.

The FI analysis does not use any intraprocedural control flow information. Instead it conservatively computes what can hold anywhere within a function, i.e., for each function, f, it uses only one alias set, $Holds_f$, to represent what may hold at any CFG node in f. Thus, in Fig. 2 the FI analysis assumes that $\langle *p, heap_{S1}\rangle$ and $\langle *p, heap_{S3}\rangle$ can flow into `f`. This results in $Holds_g = Holds_f = \{\langle *p, heap_{S1}\rangle, \langle *p, heap_{S3}\rangle, \langle *p, heap_{S4}\rangle, \langle *p, heap_{S6}\rangle, \langle *q, heap_{S7}\rangle\}$, which results in four objects, $heap_{S1}$, $heap_{S3}$, $heap_{S4}$, and $heap_{S6}$ potentially being referenced at $S8$.[4]

The FIK analysis 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.[5] For example, the precomputation will determine that all alias relations involving $*p$ on

[1] Potential calls can occur due to function pointers and virtual methods, in which the called function is not known until run time.

[2] As the PCG is not used, this can include functions that are not called.

[3] See [34, 6] for a discussion of precision trade-offs between this representation and an explicit representation, which would contain all four alias pairs.

[4] Although a final intraprocedural flow-sensitive pass can be used to improve precision [6], this pass has not been implemented.

[5] 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 [5, 6]. Only the first set is used in our example.

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 [6]

```
     T *p, *q;
     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;
S7:      q = new T;
     }
```

```
     void g(void) {
S8:      T t = *p;

     }
```

Figure 2: Example program

entry to f will be killed before the call to g at $S5$. Thus, $Holds_g$ will contain only the alias relations that are generated in f and propagated to g, i.e., $Holds_g = \{\langle *p, heap_{S4}\rangle, \langle *p, heap_{S6}\rangle \langle *q, heap_{S7}\rangle\}$. This results in two objects, $heap_{S4}$ and $heap_{S6}$, potentially being referenced at $S8$.

The FS analysis 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 FS analysis will compute $In_{S8} = \{\langle *p, heap_{S4}\rangle\}$, which is the precise solution for this simple example.

This example illustrates the theoretical precision levels of the four analyses, from FS (most precise) to AT (least precise). The AT analysis is our most efficient analysis because it is linear and only uses one set. The 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 AT analysis on the less precise/more efficient side, the FS analysis on the more precise/less efficient side, and the FI and FIK analyses in the middle.

Not studied in this paper are other flow-insensitive analyses [49, 46] that use one alias set for the whole program and limit the number of alias relations by sometimes grouping distinct variables into one named object. These analysis fall in between AT and FI in the theoretical precision spectrum.

2.2 Implementation

The analyses have been implemented in the NPIC system, an experimental program analysis system written in C++. The system uses multiple and virtual inheritance to provide an extensible framework for data flow analyses [37]. A prototype version of the IBM VisualAge C++ compiler [47, 36] is used as the front end. The abstract syntax tree constructed by the front end is transformed into a PCG and a CFG for each function, which serve as input to the alias analyses. No CFG is built for library functions. We model a call to a library function based on its semantics, thereby providing the benefits of context-sensitive analysis of such calls. 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. The FS, FI, and FIK analyses are implemented using worklists. (Section 4.2 discusses an earlier iterative implementation.) These three analyses incorporate function pointer analysis into the pointer alias analysis as described in [5, 6]. Currently, array elements and field components are not distinguished, and setjmp/longjmp[6] 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. As stated in Section 2.1, heap objects are named based on their allocation site.

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 array, used for standard I/O. The added function is similar to the one added by Ruf [40, 42] and Landi et al. [32, 30]. 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 four algorithms discussed in Section 2.1: address-taken (AT), flow-insensitive (FI), flow-insensitive with kill (FIK), and flow-sensitive (FS).

3.1 Benchmarks

Our benchmark suite contains 21 C programs, 18 provided by other researchers [32, 17, 40] and 3 from the SPEC CINT92 [4] and CINT95 [48] benchmarks.[7] 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 pointer-assignment nodes. The current 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.[8] The last two columns report the number of recursive functions (functions that are in PCG cycles) and heap allocation sites in each program. The last row of the table reports the average edge/node ratio and the average pointer-assignment node percentage, both of which are computed by averaging the corresponding values over the 21 benchmarks.

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

[7] 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 [40]. The SPEC CINT92 program 052.alvinn was named backprop in Todd Austin's benchmark suite [3]. Ruf referred to ks as part, and ft as span [42].

[8] This is more conservative than considering statements in which the left side *expression* is a pointer. Thus, 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 nodes considered during the analysis as discussed in Section 4.1.

Name	Source	LOC	Funcs	Call Sites User	Call Sites Lib	CFG Nodes	CFG Edges	$\frac{Edges}{Nodes}$	Ptr-Asgn Nodes	Rec Funcs	Allocation Sites
allroots	Landi	227	7	19	35	157	167	1.064	2.6%	2	1
052.alvinn	SPEC92	272	9	8	13	223	243	1.090	11.2%	0	0
01.qbsort	McCat	325	8	9	25	173	187	1.081	24.9%	1	5
15.trie	McCat	358	13	19	21	167	181	1.084	23.4%	3	5
04.bisect	McCat	463	9	11	18	175	189	1.080	9.7%	0	2
17.bintr	McCat	496	17	27	28	196	205	1.046	9.7%	5	1
anagram	Austin	650	16	22	38	332	365	1.099	9.0%	1	2
lex315	Landi	733	17	102	52	527	615	1.167	2.5%	0	3
ks	Austin	782	14	17	67	513	576	1.123	26.9%	0	5
05.eks	McCat	1,202	30	62	49	678	732	1.080	3.8%	0	3
08.main	McCat	1,206	41	68	53	684	700	1.023	24.3%	3	8
09.vor	McCat	1,406	52	174	28	876	917	1.047	27.5%	5	8
loader	Landi	1,539	30	79	102	687	773	1.125	8.3%	2	7
129.compress	SPEC95	1,934	25	35	28	596	652	1.094	6.4%	0	0
ft	Austin	2,156	38	63	55	732	808	1.104	19.5%	0	5
football	Landi	2,354	58	257	274	2,710	3,164	1.168	1.8%	1	0
compiler	Landi	2,360	40	349	107	1,723	2,090	1.213	1.1%	14	0
assembler	Landi	3,446	52	247	243	1,544	1,738	1.126	8.0%	0	16
yacr2	Austin	3,979	59	158	169	2,030	2,328	1.147	5.4%	5	26
simulator	Landi	4,639	111	447	226	2,686	3,123	1.163	2.7%	0	4
099.go	SPEC95	29,637	373	2,054	22	16,823	20,176	1.199	.2%	1	0
Average								1.111	10.9%		

Table 1: Static characteristics of benchmark suite

3.2 Description of Experiment

This section presents precision and efficiency results. For each benchmark and each analysis, we report the analysis time, the maximum memory used, and the average number of objects the analysis computes a dereferenced pointer can point to. The precision results for the FIK analysis are exactly the same as the FI analysis for all benchmarks. Thus, we do not explicitly include this analysis in our precision data.

The experiment was performed on a 132MHz IBM RS/6000 PowerPC 604 with 96MB RAM and 104MB paging space, running AIX 4.1.5. The executable was built with IBM's xlC compiler using the "-O3" option.

The analysis time is reported in seconds and does not include the time to build the PCG and CFGs, but does include any analysis-specific preprocessing, such as building the SEG from the CFG. This information is displayed in the top left chart in Fig. 3. The top right chart of this figure reports the high-water mark of memory usage during the analysis process. This memory usage includes the intermediate representation, the alias information, and statistics-related data. This information was obtained by using the "`ps v`" command under `AIX 4.1.5`.

To collect precision information, the system traverses the representation visiting each expression containing a pointer dereference and, using the computed alias information, reports how many named objects are aliased to the pointer expression. We report the average number of such dereferences for both reads and writes.

A pointer expression with multiple dereferences, such as `***p`, is counted as multiple dereference expressions, one for each dereference. The intermediate dereferences (`*p` and `**p`) are counted as reads. The last dereference (`***p`) is counted as a read or write depending on the context of the expression. 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, and therefore, counting it may skew results. Figure 4 classifies the type of pointer dereferenced averaged over all programs. The appendix contains a table with this information for all benchmarks.

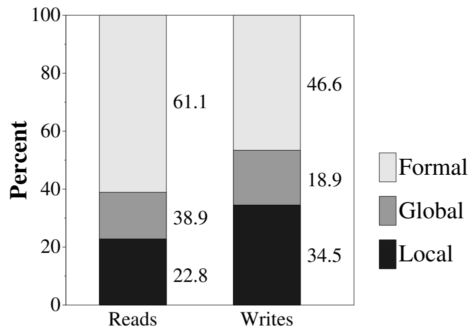


Figure 4: Classification of dereferenced pointer types for all programs

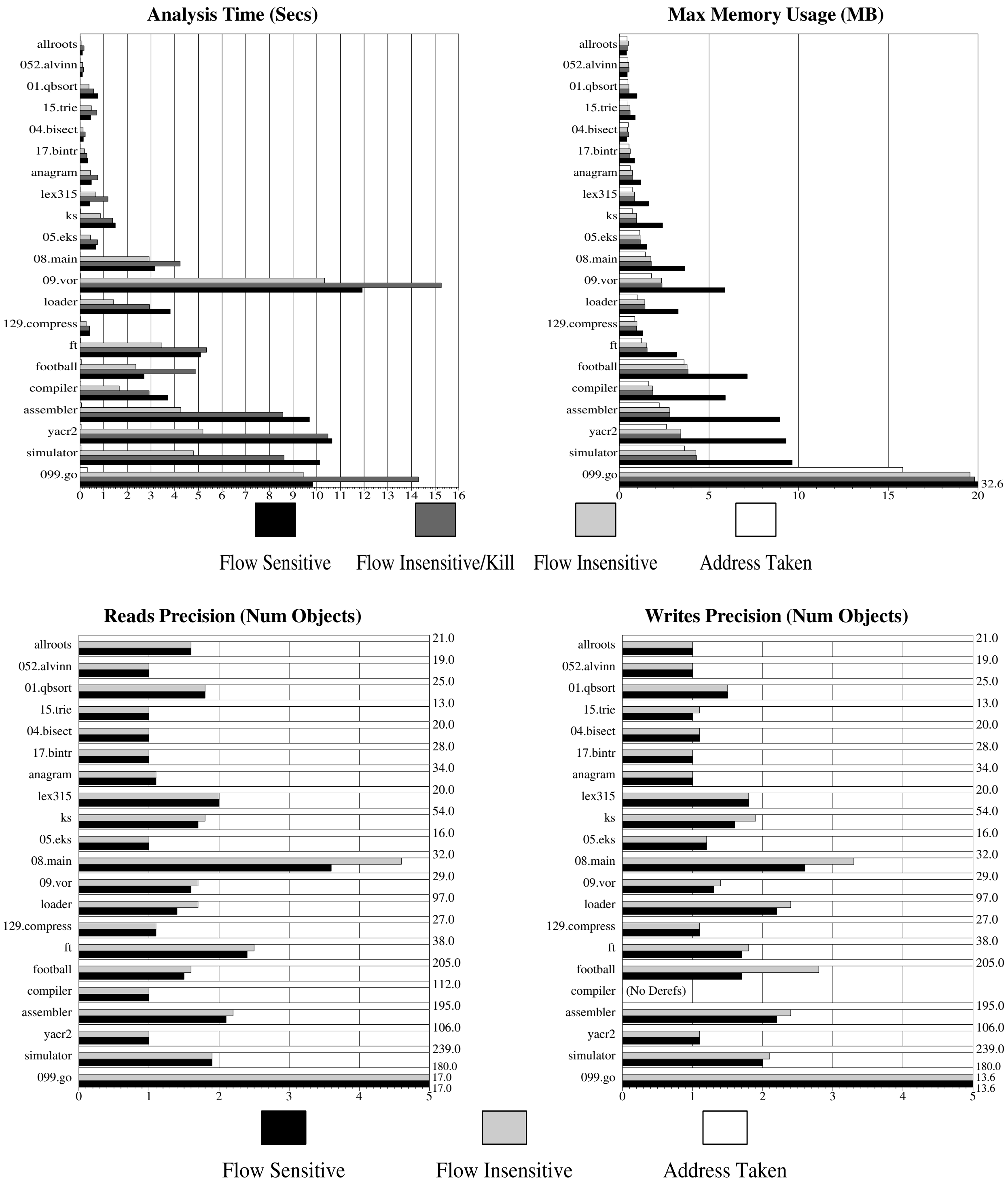


Figure 3: Analysis time, memory usage, and precision results

The manner in which the heap is modeled must be considered in evaluating precision results. 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 [40]. 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[9] 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 an imprecise algorithm, a limitation of static analysis, or a pointer dereference that corresponds to different memory locations over the program's execution.

The bottom two charts of Fig. 3 provide a graphical layout of precision information for reads and writes. Fig. 5 refines this information for the FS and FI analyses by providing a breakdown of the type of object pointed to. Fig. 6 provides similar information averaged over all programs. The charts of Fig. 7 report the percentage of dereferenced pointers that resolve to exactly one object in our model. If the object is a named variable, as opposed to a heap object, the pointer dereference could be replaced with the variable. (A table in the appendix elaborates on this information.)

3.3 Discussion

As expected, the results from the analysis speed chart of Fig. 3 indicate that the AT analysis is efficient; it takes less than .4 seconds on all programs. The FI analysis can be over twice as fast as the FS analysis, and was faster than the FS analysis in all but one program.

One would expect the memory usage for the FS analysis to be orders of magnitude larger than the FI analysis because the FS analysis can use many more alias sets.[10]. Although the initial implementation did have this property, several storage saving schemes, discussed in Section 4, significantly reduced the storage requirements and analysis time of the FS analysis. It remains to be seen if these techniques will keep the storage requirements of FS comparable with FI when larger programs are analyzed.

The precision of the AT analysis leaves much to be desired. Fig. 6 reports on average 111.9 objects for reads and 96.68 objects for writes were in the AT set.[11] As one would expect this set to increase with the size of the program, the precision for this analysis will worsen with larger programs. These results suggest that the precision of this simple analysis is not acceptable.

The results also indicate that the FIK analysis is not beneficial. On our benchmark suite it is never more precise than the FI analysis, and on some occasions requires more analysis time than the FS analysis. One explanation for the precision result may be that an alias relation created to simulate a reference parameter, in which 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 FI analysis. In 12 of the 21 benchmarks the FI analysis is as precise as the FS analysis. This seems to suggest that the added precision obtained by the FS analysis in considering control flow within a function is not significant for those benchmarks, at least where pointers are dereferenced. We offer two possible explanations:

1. Pointer variables are often not assigned more than one distinguished object within the same function. Thus, distinguishing program points within a function, a key difference between the FS and FI analyses, does not often result in an increase in precision. We have seen exceptions to this in the function `InitLists` of the `ks` benchmark and in the function `InsertPoint` in the `08.main` benchmark. In `InitLists` the same list pointer is used in two list-creating loops.

   In `InsertPoint` the same `tmp` pointer is used to traverse a list and to create a new list. In the list creation code the pointer is dereferenced to initialize the created node. The FS analysis reports that this can only be the heap object that was just created, while the FI analysis reports the possible nodes that the `tmp` pointer pointed to during the earlier list traversal. This accounts for most of the precision differences with write dereferences in the `08.main` benchmark.

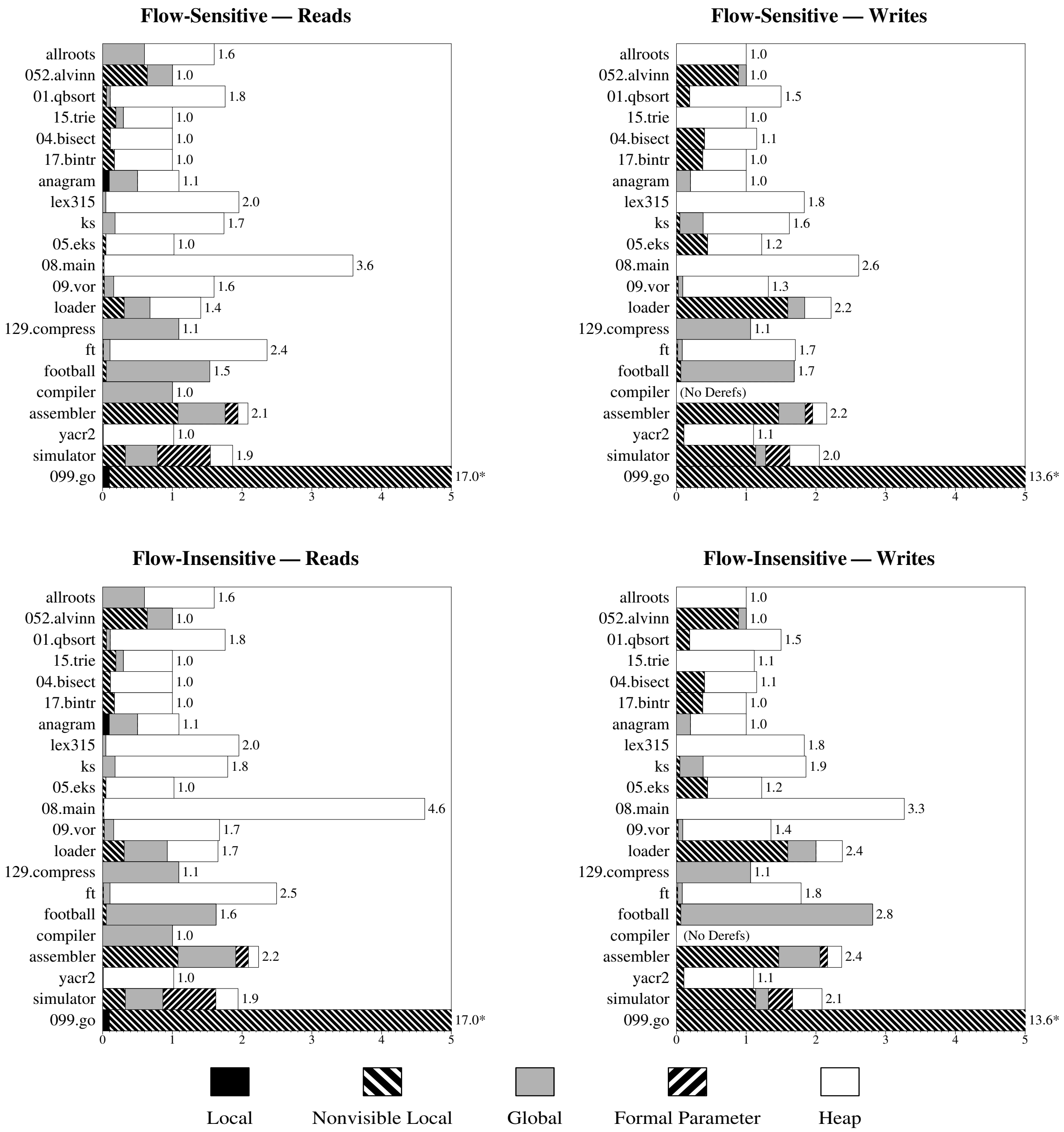
   The main cause of the differences in read dereferences in `08.main` is some peculiar code in the `main` function, in which the function `Draw_All` is called with the variable `o`, a pointer to an `object`, being passed as a parameter. In the previous statement, this variable is assigned the value of another variable which is always NULL. Thus, NULL is passed to this call. As this is the only call to this function, the FS analysis can determine that `Draw_All` and all functions it calls passing its parameter, will have NULL as the value of the pointer. However, the variable `o` is also used earlier in `main` to point to some object, which points to other objects. Thus, at the call to `Draw_All` the FI analysis does not have the benefit of the killing definition to `o` and reports that it can point to other objects besides NULL.[12]

[9] This modeling technique can increase the possibility of strong updates [9].

[10] The FS analysis *potentially* use two alias sets for every CFG node in the program plus two alias sets for each function. The FI analysis uses two alias sets for each function.

[11] The numbers differ because they are weighted with the number of reads and writes through a pointer in each program, so programs with a larger number of reads through a pointer than writes through a pointer will contribute more to the read average.

[12] The FIK analysis also suffers the same imprecision because the killing in-



$*$ The bars for 099.go are truncated. The object type breakdown for both FS and FI is

	Local	Nonvisible Local	Global	Formal Parameter	Heap
Reads	0.1	9.7	7.2	0.01	0
Writes	0.0	8.1	5.4	0.01	0

Figure 5: Breakdown of average object type pointed to by a dereferenced pointer

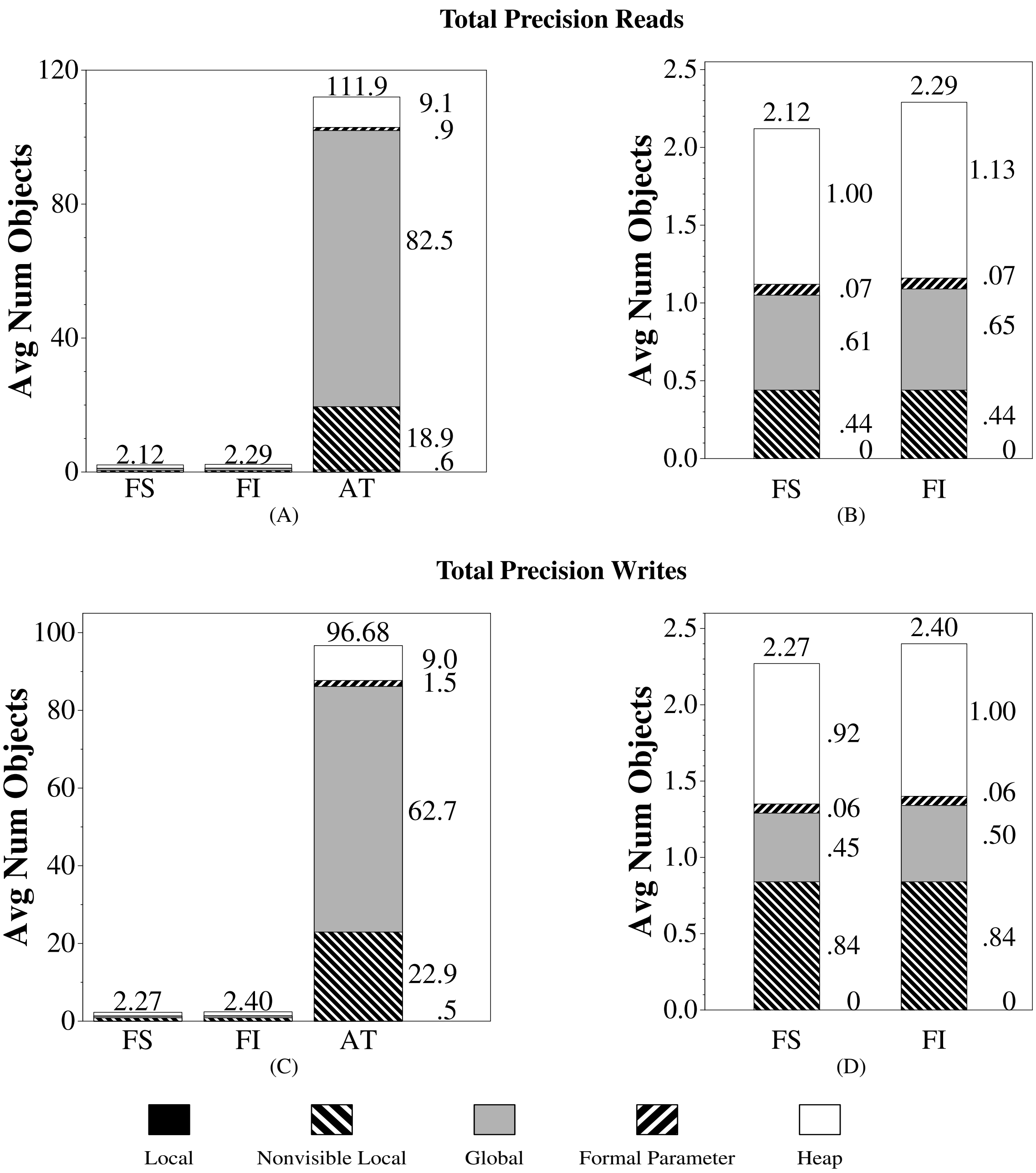


Figure 6: Charts A – D provide the average precision over all benchmarks for reads and writes. (Charts B and D do not include the AT analysis to allow the difference between the FS and FI analyses to be visible.)

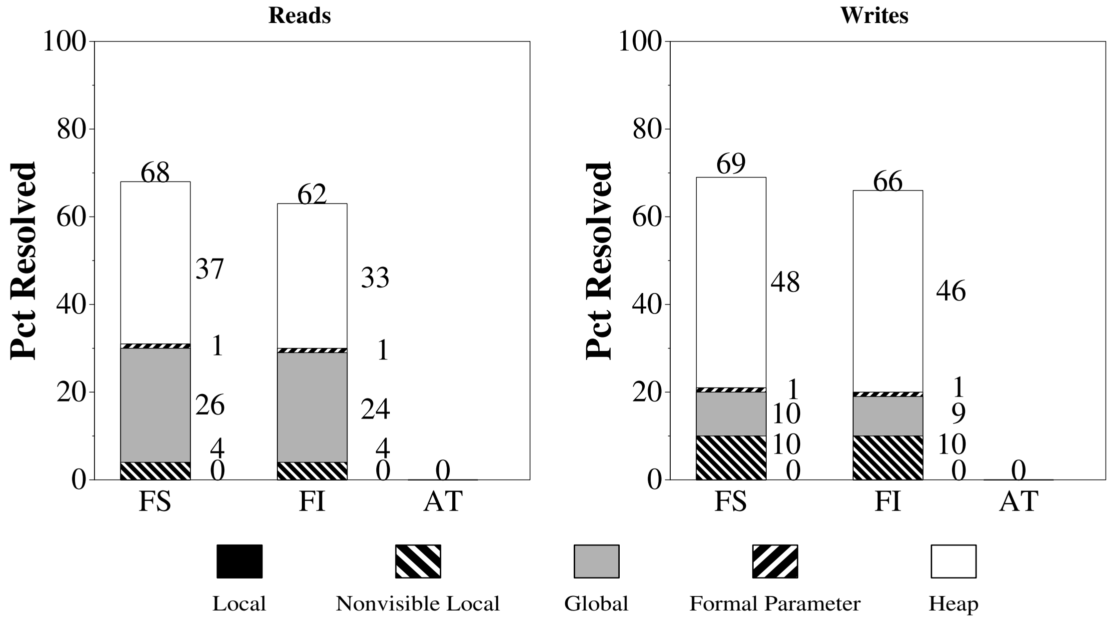


Figure 7: Percentage of dereferenced pointers that resolve to one object in our model.

Without this function call in `main` the precision for the FI analysis is much closer to the FS analysis.

2. It seems that a large number of alias relations are created at call sites because of actual/formal parameter bindings. The lack of a substantial precision difference between our FS and FI analyses may be because both algorithms rely on the same (context-insensitive) mapping mechanism at call sites.

Considering charts B and D of Fig. 3, it seems that FI is as precise as FS for pointers directed to nonvisible locals and formal parameters. Therefore, FS, if employed at all, should focus on pointers directed to the heap and global variables.

The precision results for `099.go` merit discussion. An average of 17.03 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, passing the address of 100 different actuals for the list header, resulting in 100 aliases for the formal parameter. However, because the lifetime of the formal 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 mentioned in Section 2.1.

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 2 illustrates this point for the FS analysis. For example, `099.go`, despite being our largest program and having a pointer aliased with 100 objects, is analyzed at one of the fastest rates (3,037 LOC/second, 1,724 CFG nodes/second) because these relations are not propagated throughout the program.

As suggested by Shapiro and Horwitz [45] and Diwan [15], 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.

formation it computes only removes aliases involving `o` coming from the entry set of `main` (none exist), but then adds *all* aliases generated in `main`.

3.4 Comparison with Other Results

Landi et al. [32] 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 Fig. 3. However, since their results include compiler-introduced temporaries in their "thru-deref" count [30], a direct comparison is not possible.

Stocks et al. [50] use the same metric without including temporaries for the flow-sensitive context-sensitive analysis of Landi and Ryder [31]. They report the average number of

Program	LOC	CFG Nodes	FS Time	LOC/Sec	Nodes/Sec
allroots	227	157	0.10	2,270	1,570
052.alvinn	272	223	0.09	3,022	2,478
01.qbsort	325	173	0.70	464	247
15.trie	358	167	0.44	814	380
04.bisect	463	175	0.12	3,858	1,458
17.bintr	496	196	0.30	1,653	653
anagram	650	332	0.46	1,413	722
lex315	733	527	0.39	1,879	1,351
ks	782	513	1.45	539	354
05.eks	1,202	678	0.63	1,908	1,076
08.main	1,206	684	3.03	398	226
09.vor	1,402	876	11.57	121	76
loader	1,539	687	3.60	428	191
129.compress	1,934	596	0.40	4,835	1,490
ft	2,156	732	4.68	461	156
football	2,354	2,710	2.62	898	1,034
compiler	2,360	1,723	3.53	669	488
assembler	3446	1,540	9.54	361	161
yacr2	3,979	2,030	9.86	404	206
simulator	4,639	2,684	10.02	463	268
099.go	29,637	16,822	9.76	3,037	1,724
Average	—	—	—	1,424	777

Table 2: Flow-sensitive analysis speed, computed by dividing the size of the program (as measured in LOC or CFG Nodes) by the analysis time.

objects ranges from 1.0 to 2.0 on the eight common benchmarks (`allroots`, `lex315`, `loader`, `129.compress`, `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, statements such as `cfree(TP)` located in allroots are treated as modifying the structure deallocated, and thus as a pointer dereference [30]. In fact, on the three programs in which 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 [50].

The relative precision of the flow-insensitive analysis compared to the flow-sensitive analysis is in contrast to the study of Stocks et al. [50], which compares the flow-sensitive analysis of Landi and Ryder [31] with a flow-insensitive analysis described in [54]. For the eight common benchmarks, our flow-insensitive algorithm ranges from 1.0 to 2.81 objects on average for a write dereference, compared to 1.0 to approximately 6.3 for the flow-insensitive analysis they studied. The analysis described in [54] shares the property of Steensgaard's analysis [49] in that it groups all objects pointed-to by a variable into an equivalence class. Although this can lead to spurious alias relations not present in the FI analysis, it does allow for an almost linear time algorithm, which has been shown to be fast in practice [49, 46, 55].

Emami et al. [17] report precision results for a flow-sensitive context-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 the suite they used in this study, a direct comparison with our results is not possible.

Ruf [40] 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 [42]. Since such an array will always point to the same place, the average number of objects is improved.[13] For the 11 benchmarks in common,[14] Ruf reports an overall read and write average of 1.33 and 1.38, respectively. To facilitate comparisons, we have also counted in this manner. The results for the common benchmarks are averages of 1.35 and 1.47 for the FS analysis and 1.41 and 1.54 for the FI analysis. (The results for the complete benchmark suite are contained in Tables 10 and 11 in the appendix.) We attribute the slight differences in the FS analysis to the difference in representations. As Ruf [40] 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 [45] present an empirical comparison of four flow-insensitive algorithms. Each successive algorithm can be more precise and less efficient.

- the "address-taken" analysis we included in this work,
- an almost-linear time algorithm by Steensgaard [49],
- an algorithm by Shapiro and Horwitz [46],
- an algorithm by Andersen [2]. This algorithm differs from the flow-insensitive algorithm we have studied in that it uses one alias set for the whole program rather than one set per function.

The first algorithm is the same as the AT algorithm. The remaining three algorithms can be less precise and more efficient than the algorithms studied in this paper.[15] The authors measure the precision of these analyses by implementing three dataflow analyses (GMOD, live variables, and truly live variables) and an interprocedural slicing algorithm. In addition to these alias analysis clients, the authors also report the direct precision of the alias analysis algorithms in terms of the total number of points-to relations. We agree with [17, 40] that

[13] The best illustration of this is in `099.go`, which has a large number of array references, but a low number of pointer dereferences. In this program, the average changed from 17.03 to 1.13 for reads and from 13.64 to 1.48 for writes when all uses of the "[ ]" operator were counted.

[14] The common benchmarks are `allroots`, `052.alvinn`, `anagram`, `lex315`, `ks`, `loader`, `129.compress`, `compiler`, `assembler`, `yacr2`, and `simulator`. Although [40] 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.

[15] In theory, the FI analysis can be more precise than Andersen's algorithm [2] because it considers function scopes, at the cost of using more than one alias set. However, both algorithms are likely to offer similar precision in practice because the distinguishing case is probably uncommon.

a more meaningful metric is to measure where the points-to information is used, such as where a pointer is dereferenced.

For the pointer analyses they conclude

- a more precise flow-insensitive analysis generally leads to increased precision by the subsequent analyses that use this information with varying magnitudes;
- metrics measuring the alias analysis precision tend to be good predictors on the precision of subsequent analyses that use alias information;
- more precise flow-insensitive analysis can also improve the efficiency of subsequent analyses that use this information.

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 3 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 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 nonconstant 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 = q`" or "`p->next = q->next`". [16] These figures do not include the effects of assignments of actuals to formals in order to simulate call-by-reference semantics. The column labeled "Call" is the percentage of call nodes.

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, this can result in storing redundant information. For example, node #1 in Fig. 8 does not affect any pointer value; therefore its Out set will always 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 Fig. 8 except #1 and #2. Thus, for all nodes, except #1 and #2, alias sets can be shared. We use the term *shared* 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 before performing the alias analysis. Each node that has its own set, which we call a *deep* set, is dubbed a "SEG" (*Sparse Evaluation Graph*) node.[17] 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, is summarized as follows:

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

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

Our current implementation treats all call nodes as SEG nodes.

Table 4 reports the number of alias sets before and after this optimization was applied as well as the percentage reduction. In addition to saving storage (on average 73.78% fewer alias sets were allocated), this method saves visits during the actual analysis to nodes that can not affect pointer aliasing.[18] Although we allocate fewer alias sets, the real benefit of this technique is seen during the analysis. Since we do not visit and propagate alias relations between extraneous CFG nodes, we simply have fewer alias relations being stored and copied from one CFG node to the next. This affects both the analysis's time and space use in a significant way.

Table 5 shows the effects of the efficiency improving ideas on the flow-sensitive analysis. Run times were collected as in Section 3, using the C function `clock()`, which gives the CPU time, not the elapsed time. This metric was chosen because it eliminates the effects of system load, amount of RAM vs. paging space, etc. The elapsed time for each program was approximately 2.5 times the CPU time.

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 resulting 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

[16] 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.

[17] The method described is a conservative approximation to the SEG of [11] and is similar to [39].

[18] On some benchmarks using the "No Sharing" version (column 2 in Table 4), the number of alias sets allocated is not twice the sum of nodes and functions, as would be expected (e.g. `05.eks`). Our FS analysis only allocates alias sets for CFG nodes on the first visit to the enclosing function. Since some benchmarks have unreachable functions, we do not allocate alias sets for nodes in those function's CFG, accounting for this discrepancy.

Benchmark	CFG Nodes							
		Pct	Pct	# Ptr Assign Type				Pct
	Total #	Back	Merge	1	2	3	Total Pct	Call
allroots	157	4.46%	10.83%	3	0	1	2.55%	34.39%
052.alvinn	223	13.00%	13.00%	7	14	0	9.42%	9.42%
01.qbsort	173	6.36%	12.14%	9	18	16	24.86%	19.65%
15.trie	167	0.60%	12.57%	7	12	20	23.35%	23.95%
04.bisect	175	5.71%	12.57%	4	1	9	8.00%	16.57%
17.bintr	196	4.08%	12.76%	6	6	7	9.69%	28.06%
anagram	332	4.82%	13.86%	6	15	8	8.73%	18.07%
lex315	527	2.47%	15.56%	5	2	5	2.28%	29.22%
ks	513	7.02%	12.87%	19	88	31	26.90%	16.37%
05.eks	678	9.00%	11.21%	4	13	9	3.83%	16.37%
08.main	684	4.24%	8.33%	31	54	89	25.44%	17.69%
09.vor	876	1.03%	9.93%	13	79	149	27.51%	23.06%
loader	687	3.78%	14.85%	11	20	29	8.73%	26.35%
129.compress	596	4.36%	13.42%	7	2	30	6.54%	10.57%
ft	732	4.23%	11.48%	20	68	56	19.67%	16.12%
football	2710	3.73%	12.92%	4	30	9	1.59%	19.59%
compiler	1723	1.74%	17.82%	6	14	0	1.16%	26.47%
assembler	1540	2.14%	14.03%	22	41	77	9.09%	31.82%
yacr2	2030	6.35%	15.22%	28	10	99	6.75%	16.11%
simulator	2684	1.86%	17.47%	19	16	58	3.46%	25.07%
099.go	16822	4.13%	21.49%	5	1	23	0.17%	12.34%
Averages	—	4.53%	13.54%	—	—	—	10.94%	20.82%

Table 3: 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.

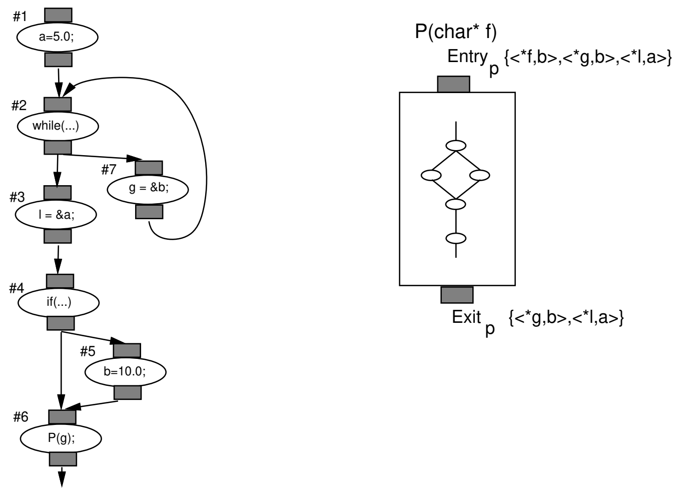


Figure 8: Example CFG (left) and procedure `P` (right). Note that `g` is a global, `a,b,l` are locals of the CFG, and `f` is a formal parameter of `P`. Possible topological numbers (ignoring the back edge $7 \rightarrow 2$) appear next to each node.

Benchmark Name	No Sharing	⟵ Sharing ⟶				Overall Speed-up
	⟵ Iterating ⟶		⟵ Worklists ⟶			
			Unsorted	Sorted		
	⟵ No Forward Bind Filter ⟶				Filter	
allroots (227 LOC)	1.41	0.54 (2.61)	0.21 (2.57)	0.24 (0.88)	0.10 (2.40)	14.10
052.alvinn (272 LOC)	1.03	0.39 (2.64)	0.09 (4.33)	0.10 (0.90)	0.10 (1.00)	10.30
01.qbsort (325 LOC)	8.22	4.16 (1.98)	1.81 (2.30)	1.19 (1.52)	0.75 (1.59)	10.96
15.trie (358 LOC)	4.59	2.26 (2.03)	0.83 (2.72)	0.72 (1.15)	0.45 (1.60)	10.20
04.bisect (463 LOC)	1.47	0.45 (3.27)	0.16 (2.81)	0.13 (1.23)	0.14 (0.93)	10.50
17.bintr (495 LOC)	2.20	1.19 (1.85)	0.33 (3.61)	0.32 (1.03)	0.32 (1.00)	6.88
anagram (650 LOC)	6.67	2.16 (3.09)	0.59 (3.66)	0.52 (1.13)	0.48 (1.08)	13.90
lex315 (733 LOC)	5.03	2.09 (2.41)	0.73 (2.86)	0.70 (1.04)	0.41 (1.71)	12.27
ks (782 LOC)	20.48	9.40 (2.18)	3.32 (2.83)	2.55 (1.30)	1.49 (1.71)	13.74
05.eks (1202 LOC)	9.21	2.87 (3.21)	0.92 (3.12)	0.87 (1.06)	0.67 (1.30)	13.75
08.main (1206 LOC)	96.37	44.80 (2.15)	18.30 (2.45)	12.39 (1.48)	3.16 (3.92)	30.50
09.vor (1406 LOC)	217.19	113.71 (1.91)	38.70 (2.94)	32.96 (1.17)	11.92 (2.77)	18.22
loader (1539 LOC)	176.49	58.37 (3.02)	26.71 (2.19)	21.20 (1.26)	3.81 (5.56)	46.32
129.compress (1934 LOC)	6.00	1.82 (3.30)	0.56 (3.25)	0.53 (1.06)	0.41 (1.29)	14.63
ft (2156 LOC)	80.94	37.07 (2.18)	14.04 (2.64)	11.25 (1.25)	5.09 (2.21)	15.90
football (2354 LOC)	275.63	61.30 (4.50)	26.87 (2.28)	22.75 (1.18)	2.70 (8.43)	102.09
compiler (2360 LOC)	39.31	12.87 (3.05)	4.19 (3.07)	4.11 (1.02)	3.70 (1.11)	10.62
assembler (3446 LOC)	668.25	240.46 (2.78)	119.31 (2.02)	97.26 (1.23)	9.69 (10.04)	68.96
yacr2 (3979 LOC)	377.11	86.50 (4.36)	26.76 (3.23)	25.93 (1.03)	10.64 (2.44)	35.44
simulator (4639 LOC)	511.49	146.35 (3.49)	82.41 (1.78)	61.84 (1.33)	10.12 (6.11)	50.54
099.go (29637 LOC)	—	228.18 —	74.01 (3.08)	65.23 (1.13)	9.83 (6.64)	23.21
Averages		2.80	2.84	1.16	3.09	25.38

Table 5: Flow-sensitive analysis run time in seconds. Numbers in parentheses are the speed-up from the previous version to the left.

Benchmark	Num. Alias Sets No Sharing	Sharing	Pct Saved
allroots	328	87	73.48%
052.alvinn	464	89	80.82%
01.qbsort	362	111	69.34%
15.trie	360	125	65.28%
04.bisect	352	78	77.84%
17.bintr	350	124	64.57%
anagram	696	159	77.16%
lex315	1088	260	76.10%
ks	1054	317	69.92%
05.eks	694	170	75.50%
08.main	1166	399	65.78%
09.vor	1842	633	65.64%
loader	1434	382	73.36%
129.compress	1152	195	83.07%
ft	1140	380	66.67%
football	5536	962	82.62%
compiler	3486	699	79.95%
assembler	3184	911	71.39%
yacr2	3960	782	80.25%
simulator	5180	1203	76.78%
099.go	—	4966	—
Average			73.78%

Table 4: Effectiveness of sharing alias sets

benefit.[19] For all our benchmarks, these averages were 10.9% and 20.8% 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, 2.80 on average. For our largest benchmark, `099.go`, the analysis did not run (due to insufficient memory) until we applied this 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" loops with the worklists — the outer loop visiting functions and the inner loop 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 function's entry set changes, it 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 4 of Table 5 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 2.84 over the iterating version, which produced an average speed-up of 2.78 over the nonshared 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 data-flow problem, it is natural to process the nodes in topological order. The next enhancement was to use priority queues (based on a topological order[20]) for the SEG and function worklists.

Consider the case of 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 Fig. 8 using the given node numbers as the topological order.) 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, except in the presence of gotos), 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 over the iterating version.

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 5 reports the analysis time using sorted worklists for both SEG nodes and functions and the previous enhancements. This enhancement resulted in an average speed-up of 1.16 over the previous version, which used nonpriority-based worklists and shared alias sets.

4.4 ForwardBind Filtering

`ForwardBind()` is a function in our analysis that propagates alias relations from call nodes to the entry set of called functions. If needed, it creates alias relations for the formals

[19] The percentage of merge nodes plays a smaller role.

[20] Topological order on a cyclic graph can be obtained by performing a depth-first search of the graph and then removing edges classified as back edges [28].

from 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. Consider those alias relations that cannot be changed or used in the called function (for example, the relation $\langle *l, a \rangle$ in Fig. 8), 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. Although correct, this is inefficient.

Our enhancement is not to propagate from the call node those alias relations that cannot be reached in 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 (distinguished objects), and associated edges (alias relations), that cannot be reached from a global or a formal of F. These removed edges (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 5 reports the effectiveness of this optimization. It provided the most dramatic speed-up, an average of 3.09 over the previous implementation, which used all other enhancements. 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.

Figure 9 shows the effects of these optimizations in a dramatic way for the `loader` benchmark.[21] We collected the data presented in this graph by repeatedly executing the "`ps v`" command (under `AIX 4.1.5`) while each of the five FS analyses was running. We recorded the `SIZE` column, which gives the virtual size of the data section of the process; this will capture all heap allocated memory usage during the analysis. The x-axis of the chart is simply marked off in samples (a sample is a single call to "`ps v`").

As the majority of our heap allocations are used to represent alias relations, the resulting memory usage can be interpreted as the number of alias relations stored by the analysis while running. The "No Sharing" version shows a characteristic curve; it grows quickly early on, but then levels off as the analysis reaches its fixed point. The difference between the No Sharing and Sharing versions shows how the large reduction in the number of alias sets reduced the number of alias relations that were stored.

The cumulative effect of all five optimizations was an average speed-up of 25.38. 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.

[21] The other benchmarks had similarly shaped graphs.

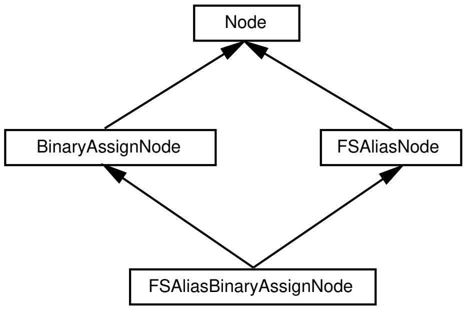


Figure 10: Sample inheritance pattern

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 10 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.

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.[22] 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 [37].

Other features of the system include

- Dakota makes extensive use of the *Abstract Factory* and *Factory Method* patterns [18] 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.

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

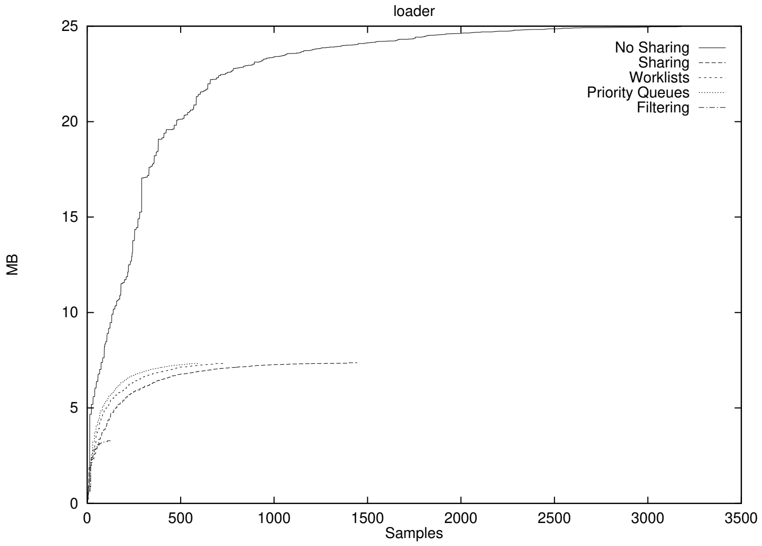


Figure 9: Memory usage over time for the `loader` benchmark.

- 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 other related work not mentioned in Section 3.4.

Ruf [40] 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 [53, 51] present an algorithm for performing context-sensitive analysis that avoids redundant analyses of functions for similar calling contexts. Ghiya and Hendren [21] present empirical data showing how points-to and connection analyses can improve traditional transformations, array dependence testing, and program understanding.

Ruf [41] describes a program partitioning technique that is used for a flow-sensitive points-to analysis, achieving a storage savings of 1.3–7.2 over existing methods. Diwan et al. [16] provide static and dynamic measurements of the effectiveness of three flow-insensitive analyses for a type-safe language (Modula-3). With the exception of AT, all three algorithms are less precise than the versions we have studied. Zhang et al. [55] report the effectiveness of applying different pointer aliasing algorithms to different parts of a program.

Hasti and Horwitz [22] present a pessimistic algorithm that attempts to increase the precision of a flow-insensitive analysis by iterating over a flow-insensitive analysis and an SSA [12] construction. No empirical results are reported. Horwitz [27] 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. [10] 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 [5]. These algorithms and other pointer aliasing techniques are further described in [6].

The Vortex [13, 8] and SUIF [52] 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

This work has described an empirical study of four pointer alias analysis algorithms that use varying degrees of flow-sensitivity. We have found that

- the address-taken analysis, although efficient, is unlikely to provide sufficient precision;
- the flow-insensitive analysis with kill is not beneficial;

- the precision of the flow-insensitive analysis is identical to that of the flow-sensitive analysis in 12 of 21 programs in our benchmark suite;
- most published implementations of flow-sensitive pointer analysis have equivalent precision.

Although the flow-sensitive analysis efficiently analyzed a program on the order of 30K LOCs, further benchmarks are needed to see if this property generalizes. We have also 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.

8 Acknowledgements

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

Bill Landi, Laurie Hendren, Erik Ruf, Barbara Ryder, 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 [10]. Discussions with Manuel Fähndrich led to reporting intermediate read dereferences, which were not considered in [24].

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

Michael Burke, Jong-Deok Choi, John Field, G. Ramalingam, Harini Srinivasan, Laureen Treacy, and the *SAS'98* program committee 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] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. Available at ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z.

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

[4] 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).

[5] 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 Workshop on Languages and Compilers for Parallel Computing.* Extended version published as Research Report RC 19546, IBM T. J. Watson Research Center, September 1994.

[6] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Interprocedural pointer alias analysis. Research Report RC 21055, IBM T. J. Watson Research Center, December 1997.

[7] Paul Carini, Michael Hind, and Harini Srinivasan. Flow-sensitive interprocedural type analysis for C++. Research Report RC 20267, IBM T. J. Watson Research Center, November 1995.

[8] 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.

[9] 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).*

[10] 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.

[11] 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.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, pages 451–490, October 1991.

[13] 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.

[14] 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)*.

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

[16] Amer Diwan, Kathryn S. McKinley, and J. Elliot B. Moss. Type-based alias analysis. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 106–117, June 1998. *SIGPLAN Notices, 33(5)*.

[17] 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)*.

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

[19] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, 1996.

[20] 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*, pages 1–15, January 1996.

[21] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *25th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 121–133, January 1998.

[22] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 97–105, June 1998. *SIGPLAN Notices, 33(5)*.

[23] 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.

[24] Michael Hind and Anthony Pioli. An empirical comparison of interprocedural pointer alias analyses. Research Report RC 21058, IBM T. J. Watson Research Center, December 1997. Also available as SUNY at New Paltz Technical Report #98-104.

[25] Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Lecture Notes in Computer Science*. Springer-Verlag, 1998. Proceedings from the *5th International Static Analysis Symposium*.

[26] 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).

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

[28] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

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

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

[31] 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).

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

[33] 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. *SIGPLAN Notices, 23(7)*.

[34] 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.

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

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

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

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

[39] 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*.

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

[41] Erik Ruf. Partitioning dataflow analyses using types. In *24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 15–26, January 1997.

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

[43] 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*, pages 16–31, January 1996.

[44] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[45] 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*.

[46] 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*, pages 1–14, January 1997.

[47] D. Soroker, M. Karasick, J. Barton, and D. Streeter. Extension mechanisms in Montana. In *8th IEEE Israeli Conference on Software and Systems*, pages 119–128, June 1997.

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

[49] Bjarne Steensgaard. Points-to analysis in almost linear time. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 32–41, January 1996.

[50] Philip A. Stocks, Barbara G. Ryder, William A. Landi, and Sean Zhang. Comparing flow and context sensitivity on the modifications-side-effects problem. In *International Symposium on Software Testing and Analysis*, pages 21–31, March 1998.

[51] Robert P. Wilson. *Efficient Context-Sensitive Pointer Analysis for C Programs*. PhD thesis, Stanford University, December 1997.

[52] 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.

[53] 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, June 1995. *SIGPLAN Notices, 30(6)*.

[54] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *4th Symposium on the Foundations of Software Engineering*, pages 81–92, October 1996.

[55] Sean Zhang, Barbara G. Ryder, and William Landi. Experiments with combined analysis for pointer aliasing. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 11–18, June 1998.

A Appendix

Tables 6 and 7 are similar to Table 3 in [17] and Fig. 4 in [40]. 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. We mark benchmarks with a "*" if there is a difference between the FS and FI analyses. (Since the precision results for the FIK analysis are exactly the same as the FI analysis for all benchmarks, we only report the FIK analysis times below the FI analysis times in parenthesis.)

Tables 8 and 9 provide alias relation characteristics about dereferenced pointers and is similar to Table 4 in [17]. The first set of columns from "Local" to "FP" classifies the type of variable that is dereferenced, providing more detailed information than Fig. 4. 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. 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 column labelled"Local" corresponds to locals in the same function in which the dereference occurred. The column labelled "NV" (nonvisible) corresponds to local variables of a function other than the one where the dereference occurred. The number of heap objects may vary if an alternative heap naming scheme is used.[23]

[23] The last column of these tables should correspond to the "Avg" column of Tables 6 and 7. However, in `08.main`, there are eight pointer dereferences

Tables 10 and 11 report precision results similar in the manner of Tables 6 and 7. The only difference is that these tables count all uses of "$*$", "$->$", "[ ]" as dereferences as is done in [40]. Once again, the precision numbers for the flow-insensitive analysis with kill are identical to the version without kill and are omitted.

that, if executed, would dereference the NULL pointer. Only the FS analysis determines this fact. In Tables 6 and 7 these dereferences were averaged in as pointing to zero objects for the FS analysis. However, in the second half of Tables 8 and 9 the objects are classified into their type. Since there is no type associated with "no object", we do not include these eight dereferences in the average given for the FS analysis, resulting in a higher average. Pointers to a FILE struct (as returned by `fopen`) are treated as a global.

Program	Analysis	Time	Type	Total Derefs	Num Objects							
					Avg	1	2	3	>3	Max	Med	
allroots	FS	0.1	read	40	1.60	16	24	0	0	2	2.0	
			write	2	1.00	2	0	0	0	1	1.0	
(7 Functions)	FI	0.1	read		1.60	16	24	0	0	2	2.0	
		(0.2)	write		1.00	2	0	0	0	1	1.0	
(227 LOC)	AT	0.0	read		21.00	0	0	0	40	21	21.0	
			write		21.00	0	0	0	2	21	21.0	
052.alvinn	FS	0.1	read	25	1.00	25	0	0	0	1	1.0	
			write	9	1.00	9	0	0	0	1	1.0	
(9 Functions)	FI	0.1	read		1.00	25	0	0	0	1	1.0	
		(0.2)	write		1.00	9	0	0	0	1	1.0	
(272 LOC)	AT	0.0	read		19.00	0	0	0	25	19	19.0	
			write		19.00	0	0	0	9	19	19.0	
01.qbsort	FS	0.8	read	37	1.76	23	0	14	0	3	1.0	
			write	16	1.50	10	4	2	0	3	1.0	
(7 Functions)	FI	0.4	read		1.76	23	0	14	0	3	1.0	
		(0.6)	write		1.50	10	4	2	0	3	1.0	
(325 LOC)	AT	0.0	read		25.00	0	0	0	37	25	25.0	
			write		25.00	0	0	0	16	25	25.0	
15.trie	FS	0.4	read	37	1.00	37	0	0	0	1	1.0	
			write	26	1.00	26	0	0	0	1	1.0	*
(12 Functions)	FI	0.5	read		1.00	37	0	0	0	1	1.0	
		(0.7)	write		1.12	23	3	0	0	2	1.0	
(358 LOC)	AT	0.0	read		13.00	0	0	0	37	13	13.0	
			write		13.00	0	0	0	26	13	13.0	
04.bisect	FS	0.1	read	27	1.00	27	0	0	0	1	1.0	
			write	20	1.15	19	0	0	1	4	1.0	
(9 Functions)	FI	0.1	read		1.00	27	0	0	0	1	1.0	
		(0.2)	write		1.15	19	0	0	1	4	1.0	
(463 LOC)	AT	0.0	read		20.00	0	0	0	27	20	20.0	
			write		20.00	0	0	0	20	20	20.0	
17.bintr	FS	0.3	read	24	1.00	24	0	0	0	1	1.0	
			write	8	1.00	8	0	0	0	1	1.0	
(16 Functions)	FI	0.2	read		1.00	24	0	0	0	1	1.0	
		(0.3)	write		1.00	8	0	0	0	1	1.0	
(496 LOC)	AT	0.0	read		28.00	0	0	0	24	28	28.0	
			write		28.00	0	0	0	8	28	28.0	
anagram	FS	0.5	read	32	1.09	29	3	0	0	2	1.0	
			write	10	1.00	10	0	0	0	1	1.0	
(16 Functions)	FI	0.4	read		1.09	29	3	0	0	2	1.0	
		(0.8)	write		1.00	10	0	0	0	1	1.0	
(650 LOC)	AT	0.0	read		34.00	0	0	0	32	34	34.0	
			write		34.00	0	0	0	10	34	34.0	
lex315	FS	0.4	read	22	1.95	1	21	0	0	2	2.0	
			write	6	1.83	1	5	0	0	2	2.0	
(17 Functions)	FI	0.7	read		1.95	1	21	0	0	2	2.0	
		(1.2)	write		1.83	1	5	0	0	2	2.0	
(733 LOC)	AT	0.0	read		20.00	0	0	0	22	20	20.0	
			write		20.00	0	0	0	6	20	20.0	
ks	FS	1.5	read	73	1.74	19	54	0	0	2	2.0	*
			write	42	1.62	16	26	0	0	2	2.0	*
(14 Functions)	FI	0.9	read		1.79	15	58	0	0	2	2.0	
		(1.4)	write		1.86	6	36	0	0	2	2.0	
(782 LOC)	AT	0.0	read		54.00	0	0	0	73	54	54.0	
			write		54.00	0	0	0	42	54	54.0	
05.eks	FS	0.7	read	88	1.02	86	2	0	0	2	1.0	
			write	27	1.22	21	6	0	0	2	1.0	
(29 Functions)	FI	0.4	read		1.02	86	2	0	0	2	1.0	
		(0.7)	write		1.22	21	6	0	0	2	1.0	
(1202 LOC)	AT	0.0	read		16.00	0	0	0	88	16	16.0	
			write		16.00	0	0	0	27	16	16.0	
08.main	FS	3.2	read	218	3.45	94	7	0	109	6	5.0	*
			write	75	2.61	44	5	0	26	6	1.0	*
(41 Functions)	FI	2.9	read		4.62	29	39	0	150	6	6.0	
		(4.2)	write		3.27	41	0	0	34	6	1.0	
(1206 LOC)	AT	0.0	read		32.00	0	0	0	279	32	32.0	
			write		32.00	0	0	0	96	32	32.0	

Table 6: Part 1 of 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	Analysis	Time	Type	Tot Derefs	Num Objects Avg	1	2	3	>3	Max	Med	
09.vor	FS	11.9	read	321	1.60	160	130	31	0	3	2.0	*
			write	158	1.32	119	28	11	0	3	1.0	*
(52 Functions)	FI	10.3	read		1.68	135	155	31	0	3	2.0	
		(15.3)	write		1.35	113	34	11	0	3	1.0	
(1402 LOC)	AT	0.0	read		29.00	0	0	0	321	29	29.0	
			write		29.00	0	0	0	158	29	29.0	
loader	FS	3.8	read	81	1.41	53	27	0	1	7	1.0	*
			write	37	2.22	24	7	1	5	9	1.0	*
(30 Functions)	FI	1.4	read		1.65	51	23	0	7	7	1.0	
		(2.9)	write		2.38	24	5	1	7	9	1.0	
(1539 LOC)	AT	0.0	read		97.00	0	0	0	87	97	97.0	
			write		97.00	0	0	0	37	97	97.0	
129.compress	FS	0.4	read	22	1.09	20	2	0	0	2	1.0	
			write	33	1.06	31	2	0	0	2	1.0	
(25 Functions)	FI	0.3	read		1.09	20	2	0	0	2	1.0	
		(0.4)	write		1.06	31	2	0	0	2	1.0	
(1934 LOC)	AT	0.0	read		27.00	0	0	0	22	27	27.0	
			write		27.00	0	0	0	33	27	27.0	
ft	FS	5.1	read	95	2.36	25	11	59	0	3	3.0	*
			write	61	1.70	38	3	20	0	3	1.0	*
(38 Functions)	FI	3.5	read		2.49	24	0	71	0	3	3.0	
		(5.3)	write		1.79	37	0	24	0	3	1.0	
(2156 LOC)	AT	0.0	read		38.00	0	0	0	95	38	38.0	
			write		38.00	0	0	0	61	38	38.0	
football	FS	2.7	read	367	1.54	269	5	91	2	8	1.0	*
			write	16	1.69	10	1	5	0	3	1.0	*
(58 Functions)	FI	2.4	read		1.63	259	4	91	13	8	1.0	
		(4.9)	write		2.81	4	1	5	6	4	3.0	
(2354 LOC)	AT	0.1	read		205.00	0	0	0	395	205	205.0	
			write		205.00	0	0	0	18	205	205.0	
compiler	FS	3.7	read	18	1.00	18	0	0	0	1	1.0	
			write	0	0.00	0	0	0	0	0	0.0	
(40 Functions)	FI	1.7	read		1.00	18	0	0	0	1	1.0	
		(2.9)	write		0.00	0	0	0	0	0	0.0	
(2360 LOC)	AT	0.0	read		112.00	0	0	0	18	112	112.0	
			write		0.00	0	0	0	0	0	0.0	
assembler	FS	9.7	read	263	2.08	183	23	0	57	9	1.0	*
			write	103	2.16	62	16	0	25	9	1.0	*
(52 Functions)	FI	4.3	read		2.24	176	19	0	68	9	1.0	
		(8.6)	write		2.37	58	14	0	31	9	1.0	
(3446 LOC)	AT	0.1	read		195.00	0	0	0	268	195	195.0	
			write		195.00	0	0	0	103	195	195.0	
yacr2	FS	10.6	read	360	1.02	353	7	0	0	2	1.0	
			write	114	1.11	103	10	1	0	3	1.0	
(59 Functions)	FI	5.2	read		1.02	353	7	0	0	2	1.0	
		(10.5)	write		1.11	103	10	1	0	3	1.0	
(3979 LOC)	AT	0.1	read		106.00	0	0	0	360	106	106.0	
			write		106.00	0	0	0	114	106	106.0	
simulator	FS	10.1	read	154	1.86	111	5	28	10	13	1.0	*
			write	104	2.05	67	11	16	10	13	1.0	*
(111 Functions)	FI	4.8	read		1.94	109	1	30	14	13	1.0	
		(8.6)	write		2.09	67	9	16	12	13	1.0	
(4639 LOC)	AT	0.1	read		239.00	0	0	0	157	239	239.0	
			write		239.00	0	0	0	104	239	239.0	
099.go	FS	9.8	read	62	17.03	16	12	3	31	100	4.0	
			write	47	13.64	6	9	9	23	100	3.0	
(373 Functions)	FI	9.4	read		17.03	16	12	3	31	100	4.0	
		(14.3)	write		13.64	6	9	9	23	100	3.0	
(29637 LOC)	AT	0.3	read		180.00	0	0	0	67	180	180.0	
			write		180.00	0	0	0	47	180	180.0	
Total	FS	—	read	2366	2.11	1589	333	226	210	100	—	
		—	write	914	2.26	626	133	65	90	100	—	
	FI	—	read		2.28	1473	370	240	283	100	—	
		—	write		2.40	593	138	69	114	100	—	
	AT	—	read		106.81	0	0	0	2366	239	—	
		—	write		94.30	0	0	0	914	239	—	

Table 7: Part 2 of efficiency/precision measurements: numbers in “()” are the FIK analysis times. An entry with a “∗” indicates an increase in precision for the flow-sensitive analysis.

		Dereferenced Pointer Type					Average Object Type					
Program	Type	Local	Global	FP	Total	Analysis	Local	NV	Global	FP	Heap	Total
allroots	read	16	0	24	40	FS	0.00	0.00	0.60	0.00	1.00	1.60
						FI	0.00	0.00	0.60	0.00	1.00	1.60
						AT	0.00	0.00	20.00	0.00	1.00	21.00
1 heap object	write	2	0	0	2	FS	0.00	0.00	0.00	0.00	1.00	1.00
						FI	0.00	0.00	0.00	0.00	1.00	1.00
						AT	0.00	0.00	20.00	0.00	1.00	21.00
052.alvinn	read	13	0	12	25	FS	0.00	0.64	0.36	0.00	0.00	1.00
						FI	0.00	0.64	0.36	0.00	0.00	1.00
						AT	0.00	3.00	16.00	0.00	0.00	19.00
0 heap objects	write	7	0	2	9	FS	0.00	0.89	0.11	0.00	0.00	1.00
						FI	0.00	0.89	0.11	0.00	0.00	1.00
						AT	0.00	3.00	16.00	0.00	0.00	19.00
01.qbsort	read	10	0	27	37	FS	0.00	0.05	0.05	0.00	1.65	1.76
						FI	0.00	0.05	0.05	0.00	1.65	1.76
						AT	0.27	3.73	16.00	0.00	5.00	25.00
5 heap objects	write	11	0	5	16	FS	0.00	0.19	0.00	0.00	1.31	1.50
						FI	0.00	0.19	0.00	0.00	1.31	1.50
						AT	1.00	3.00	16.00	0.00	5.00	25.00
15.trie	read	1	0	36	37	FS	0.00	0.19	0.11	0.00	0.70	1.00
						FI	0.00	0.19	0.11	0.00	0.70	1.00
						AT	0.11	0.89	7.00	0.00	5.00	13.00
5 heap objects	write	11	0	15	26	FS	0.00	0.00	0.00	0.00	1.00	1.00
						FI	0.00	0.00	0.00	0.00	1.12	1.12
						AT	0.00	1.00	7.00	0.00	5.00	13.00
04.bisect	read	5	0	22	27	FS	0.00	0.11	0.00	0.00	0.89	1.00
						FI	0.00	0.11	0.00	0.00	0.89	1.00
						AT	1.00	8.00	9.00	0.00	2.00	20.00
2 heap objects	write	7	0	13	20	FS	0.00	0.40	0.00	0.00	0.75	1.15
						FI	0.00	0.40	0.00	0.00	0.75	1.15
						AT	1.80	7.20	9.00	0.00	2.00	20.00
17.bintr	read	6	0	18	24	FS	0.00	0.17	0.00	0.00	0.83	1.00
						FI	0.00	0.17	0.00	0.00	0.83	1.00
						AT	0.04	4.96	22.00	0.00	1.00	28.00
1 heap object	write	5	0	3	8	FS	0.00	0.38	0.00	0.00	0.62	1.00
						FI	0.00	0.38	0.00	0.00	0.62	1.00
						AT	0.12	4.88	22.00	0.00	1.00	28.00
anagram	read	19	2	11	32	FS	0.09	0.00	0.41	0.00	0.59	1.09
						FI	0.09	0.00	0.41	0.00	0.59	1.09
						AT	0.56	2.44	29.00	0.00	2.00	34.00
2 heap objects	write	9	0	1	10	FS	0.00	0.00	0.20	0.00	0.80	1.00
						FI	0.00	0.00	0.20	0.00	0.80	1.00
						AT	1.00	2.00	29.00	0.00	2.00	34.00
lex315	read	0	3	19	22	FS	0.00	0.00	0.05	0.00	1.91	1.95
						FI	0.00	0.00	0.05	0.00	1.91	1.95
						AT	0.00	0.00	17.00	0.00	3.00	20.00
3 heap objects	write	1	4	1	6	FS	0.00	0.00	0.00	0.00	1.83	1.83
						FI	0.00	0.00	0.00	0.00	1.83	1.83
						AT	0.00	0.00	17.00	0.00	3.00	20.00
ks	read	59	2	12	73	FS	0.00	0.00	0.18	0.00	1.56	1.74
						FI	0.00	0.00	0.18	0.00	1.62	1.79
						AT	0.04	1.96	47.00	0.00	5.00	54.00
5 heap objects	write	25	2	15	42	FS	0.00	0.05	0.33	0.00	1.24	1.62
						FI	0.00	0.05	0.33	0.00	1.48	1.86
						AT	0.17	1.83	47.00	0.00	5.00	54.00
05.eks	read	6	6	76	88	FS	0.00	0.05	0.00	0.00	0.98	1.02
						FI	0.00	0.05	0.00	0.00	0.98	1.02
						AT	1.27	4.73	7.00	0.00	3.00	16.00
3 heap objects	write	6	0	21	27	FS	0.00	0.44	0.00	0.00	0.78	1.22
						FI	0.00	0.44	0.00	0.00	0.78	1.22
						AT	0.52	5.48	7.00	0.00	3.00	16.00
08.main	read	112	0	98	210	FS	0.00	0.02	0.00	0.00	3.57	3.59
						FI	0.00	0.02	0.00	0.00	4.60	4.62
						AT	0.00	2.00	22.00	0.00	8.00	32.00
8 heap objects	write	54	0	21	75	FS	0.00	0.00	0.00	0.00	2.61	2.61
						FI	0.00	0.00	0.00	0.00	3.27	3.27
						AT	0.00	2.00	22.00	0.00	8.00	32.00

Table 8: Dereferenced pointer characteristics (FP = formal parameter, NV = nonvisible local)

Program	Type	Dereferenced Pointer Type				Analysis	Average Object Type					
		Local	Global	FP	Total		Local	NV	Global	FP	Heap	Total
09.vor	read	107	31	183	321	FS	0.00	0.02	0.14	0.00	1.44	1.60
						FI	0.00	0.02	0.14	0.00	1.52	1.68
						AT	0.10	5.90	15.00	0.00	8.00	29.00
8 heap objects	write	70	37	51	158	FS	0.00	0.03	0.06	0.00	1.23	1.32
						FI	0.00	0.03	0.06	0.00	1.27	1.35
						AT	0.13	5.87	15.00	0.00	8.00	29.00
loader	read	22	1	58	81	FS	0.00	0.31	0.37	0.00	0.73	1.41
						FI	0.00	0.31	0.62	0.00	0.73	1.65
						AT	2.97	27.03	60.00	0.00	7.00	97.00
7 heap objects	write	4	8	25	37	FS	0.00	1.60	0.24	0.00	0.38	2.22
						FI	0.00	1.60	0.41	0.00	0.38	2.38
						AT	1.57	28.43	60.00	0.00	7.00	97.00
129.compress	read	6	2	14	22	FS	0.00	0.00	1.09	0.00	0.00	1.09
						FI	0.00	0.00	1.09	0.00	0.00	1.09
						AT	0.00	2.00	25.00	0.00	0.00	27.00
0 heap objects	write	23	2	8	33	FS	0.00	0.00	1.06	0.00	0.00	1.06
						FI	0.00	0.00	1.06	0.00	0.00	1.06
						AT	0.00	2.00	25.00	0.00	0.00	27.00
ft	read	44	9	42	95	FS	0.00	0.01	0.09	0.00	2.25	2.36
						FI	0.00	0.01	0.09	0.00	2.39	2.49
						AT	0.12	0.88	32.00	0.00	5.00	38.00
5 heap objects	write	44	4	13	61	FS	0.00	0.02	0.07	0.00	1.62	1.70
						FI	0.00	0.02	0.07	0.00	1.71	1.79
						AT	0.08	0.92	32.00	0.00	5.00	38.00
football	read	12	1	354	367	FS	0.00	0.05	1.49	0.00	0.00	1.54
						FI	0.00	0.05	1.58	0.00	0.00	1.63
						AT	0.21	7.79	197.00	0.00	0.00	205.00
0 heap objects	write	4	0	12	16	FS	0.00	0.06	1.62	0.00	0.00	1.69
						FI	0.00	0.06	2.75	0.00	0.00	2.81
						AT	0.56	7.44	197.00	0.00	0.00	205.00
compiler	read	5	13	0	18	FS	0.00	0.00	1.00	0.00	0.00	1.00
						FI	0.00	0.00	1.00	0.00	0.00	1.00
						AT	0.67	2.33	109.00	0.00	0.00	112.00
0 heap objects	write	0	0	0	0	FS	0.00	0.00	0.00	0.00	0.00	0.00
						FI	0.00	0.00	0.00	0.00	0.00	0.00
						AT	0.00	0.00	0.00	0.00	0.00	0.00
assembler	read	70	1	192	263	FS	0.00	1.08	0.68	0.18	0.14	2.08
						FI	0.00	1.08	0.83	0.18	0.14	2.24
						AT	2.14	50.86	125.00	1.00	16.00	195.00
16 heap objects	write	16	3	84	103	FS	0.00	1.47	0.38	0.11	0.20	2.16
						FI	0.00	1.47	0.59	0.11	0.20	2.37
						AT	1.67	51.33	125.00	1.00	16.00	195.00
yacr2	read	5	271	84	360	FS	0.00	0.01	0.00	0.00	1.01	1.02
						FI	0.00	0.01	0.00	0.00	1.01	1.02
						AT	0.12	15.88	64.00	0.00	26.00	106.00
26 heap objects	write	8	77	29	114	FS	0.00	0.11	0.00	0.00	1.00	1.11
						FI	0.00	0.11	0.00	0.00	1.00	1.11
						AT	0.36	15.64	64.00	0.00	26.00	106.00
simulator	read	13	38	103	154	FS	0.00	0.32	0.46	0.76	0.32	1.86
						FI	0.00	0.32	0.54	0.76	0.32	1.94
						AT	0.99	57.01	165.00	12.00	4.00	239.00
4 heap objects	write	8	36	60	104	FS	0.00	1.13	0.14	0.35	0.42	2.05
						FI	0.00	1.13	0.18	0.35	0.42	2.09
						AT	0.86	57.14	165.00	12.00	4.00	239.00
099.go	read	6	0	56	62	FS	0.10	9.68	7.16	0.10	0.00	17.03
						FI	0.10	9.68	7.16	0.10	0.00	17.03
						AT	0.19	105.80	73.00	1.00	0.00	180.00
0 heap objects	write	0	0	47	47	FS	0.00	8.15	5.40	0.09	0.00	13.64
						FI	0.00	8.15	5.40	0.09	0.00	13.64
						AT	0.06	105.90	73.00	1.00	0.00	180.00
Total	read	22.8%	16.1%	61.1%	2358	FS	0.00	0.44	0.61	0.07	1.00	1.69
						FI	0.00	0.44	0.65	0.07	1.13	1.86
						AT	0.58	18.93	82.47	0.94	9.08	93.06
	write	34.5%	18.9%	46.6%	914	FS	0.00	0.84	0.45	0.06	0.92	1.43
						FI	0.00	0.84	0.50	0.06	1.00	1.56
						AT	0.53	22.92	62.71	1.53	8.99	73.76

Table 9: Dereferenced pointer characteristics (FP = formal parameter, NV = nonvisible local)

Program	Analysis	Type	Tot Derefs	Num Objects							
				Avg	1	2	3	>3	Max	Med	
allroots	FS	read	43	1.56	19	24	0	0	2	2.0	
		write	4	1.00	4	0	0	0	1	1.0	
(7 Functions)	FI	read		1.56	19	24	0	0	2	2.0	
		write		1.00	4	0	0	0	1	1.0	
(227 LOC)	AT	read		21.00	0	0	0	43	21	21.0	
		write		21.00	0	0	0	4	21	21.0	
052.alvinn	FS	read	59	1.00	59	0	0	0	1	1.0	
		write	22	1.00	22	0	0	0	1	1.0	
(9 Functions)	FI	read		1.00	59	0	0	0	1	1.0	
		write		1.00	22	0	0	0	1	1.0	
(272 LOC)	AT	read		19.00	0	0	0	59	19	19.0	
		write		19.00	0	0	0	22	19	19.0	
01.qbsort	FS	read	37	1.76	23	0	14	0	3	1.0	
		write	18	1.44	12	4	2	0	3	1.0	
(7 Functions)	FI	read		1.76	23	0	14	0	3	1.0	
		write		1.44	12	4	2	0	3	1.0	
(325 LOC)	AT	read		25.00	0	0	0	37	25	25.0	
		write		25.00	0	0	0	18	25	25.0	
15.trie	FS	read	37	1.00	37	0	0	0	1	1.0	
		write	28	1.00	28	0	0	0	1	1.0	*
(12 Functions)	FI	read		1.00	37	0	0	0	1	1.0	
		write		1.11	25	3	0	0	2	1.0	
(358 LOC)	AT	read		13.00	0	0	0	37	13	13.0	
		write		13.00	0	0	0	28	13	13.0	
04.bisect	FS	read	27	1.00	27	0	0	0	1	1.0	
		write	22	1.14	21	0	0	1	4	1.0	
(9 Functions)	FI	read		1.00	27	0	0	0	1	1.0	
		write		1.14	21	0	0	1	4	1.0	
(463 LOC)	AT	read		20.00	0	0	0	27	20	20.0	
		write		20.00	0	0	0	22	20	20.0	
17.bintr	FS	read	27	1.00	27	0	0	0	1	1.0	
		write	11	1.00	11	0	0	0	1	1.0	
(16 Functions)	FI	read		1.00	27	0	0	0	1	1.0	
		write		1.00	11	0	0	0	1	1.0	
(496 LOC)	AT	read		28.00	0	0	0	27	28	28.0	
		write		28.00	0	0	0	11	28	28.0	
anagram	FS	read	64	1.05	61	3	0	0	2	1.0	
		write	28	1.00	28	0	0	0	1	1.0	
(16 Functions)	FI	read		1.05	61	3	0	0	2	1.0	
		write		1.00	28	0	0	0	1	1.0	
(650 LOC)	AT	read		34.00	0	0	0	64	34	34.0	
		write		34.00	0	0	0	28	34	34.0	
lex315	FS	read	27	1.78	6	21	0	0	2	2.0	
		write	10	1.50	5	5	0	0	2	1.5	
(17 Functions)	FI	read		1.78	6	21	0	0	2	2.0	
		write		1.50	5	5	0	0	2	1.5	
(733 LOC)	AT	read		20.00	0	0	0	27	20	20.0	
		write		20.00	0	0	0	10	20	20.0	
ks	FS	read	116	1.47	62	54	0	0	2	1.0	*
		write	66	1.39	40	26	0	0	2	1.0	*
(14 Functions)	FI	read		1.56	51	65	0	0	2	2.0	
		write		1.56	29	37	0	0	2	2.0	
(782 LOC)	AT	read		54.00	0	0	0	116	54	54.0	
		write		54.00	0	0	0	66	54	54.0	
05.eks	FS	read	88	1.02	86	2	0	0	2	1.0	
		write	29	1.21	23	6	0	0	2	1.0	
(29 Functions)	FI	read		1.02	86	2	0	0	2	1.0	
		write		1.21	23	6	0	0	2	1.0	
(1202 LOC)	AT	read		16.00	0	0	0	88	16	16.0	
		write		16.00	0	0	0	29	16	16.0	
08.main	FS	read	218	3.46	94	7	0	109	6	5.0	*
		write	89	2.36	58	5	0	26	6	1.0	*
(41 Functions)	FI	read		4.62	29	39	0	150	6	6.0	
		write		2.91	55	0	0	34	6	1.0	
(1206 LOC)	AT	read		32.00	0	0	0	218	32	32.0	
		write		32.00	0	0	0	89	32	32.0	

Table 10: Part 1 of precision measurements counting all uses of "$*$", "$->$", "[ ]". An entry with a "$*$" indicates an increase in precision for the flow-sensitive analysis.

			Tot	Num Objects							
Program	Analysis	Type	Derefs	Avg	1	2	3	>3	Max	Med	
09.vor	FS	read	321	1.60	160	130	31	0	3	2.0	*
		write	160	1.31	121	28	11	0	3	1.0	*
(52 Functions)	FI	read		1.68	135	155	31	0	3	2.0	
		write		1.35	115	34	11	0	3	1.0	
(1402 LOC)	AT	read		29.00	0	0	0	321	29	29.0	
		write		29.00	0	0	0	160	29	29.0	
loader	FS	read	84	1.39	56	27	0	1	7	1.0	*
		write	47	1.96	34	7	1	5	9	1.0	*
(30 Functions)	FI	read		1.63	54	23	0	7	7	1.0	
		write		2.09	34	5	1	7	9	1.0	
(1539 LOC)	AT	read		97.00	0	0	0	84	97	97.0	
		write		97.00	0	0	0	47	97	97.0	
129.compress	FS	read	54	1.04	52	2	0	0	2	1.0	
		write	44	1.05	42	2	0	0	2	1.0	
(25 Functions)	FI	read		1.04	52	2	0	0	2	1.0	
		write		1.05	42	2	0	0	2	1.0	
(1934 LOC)	AT	read		27.00	0	0	0	54	27	27.0	
		write		27.00	0	0	0	44	27	27.0	
ft	FS	read	111	2.45	25	11	75	0	3	3.0	*
		write	70	1.79	41	3	26	0	3	1.0	*
(38 Functions)	FI	read		2.57	24	0	87	0	3	3.0	
		write		1.89	39	0	31	0	3	1.0	
(2156 LOC)	AT	read		38.00	0	0	0	111	38	38.0	
		write		38.00	0	0	0	70	38	38.0	
football	FS	read	2194	1.09	2096	5	91	2	8	1.0	*
		write	201	1.05	195	1	5	0	3	1.0	*
(58 Functions)	FI	read		1.10	2086	4	91	13	8	1.0	
		write		1.14	189	1	5	6	4	1.0	
(2354 LOC)	AT	read		205.00	0	0	0	2194	205	205.0	
		write		205.00	0	0	0	201	205	205.0	
compiler	FS	read	110	1.00	110	0	0	0	1	1.0	
		write	51	1.00	51	0	0	0	1	1.0	
(40 Functions)	FI	read		1.00	110	0	0	0	1	1.0	
		write		1.00	51	0	0	0	1	1.0	
(2360 LOC)	AT	read		112.00	0	0	0	110	112	112.0	
		write		112.00	0	0	0	51	112	112.0	
assembler	FS	read	265	2.08	185	23	0	57	9	1.0	*
		write	121	1.98	80	16	0	25	9	1.0	*
(52 Functions)	FI	read		2.23	178	19	0	68	9	1.0	
		write		2.17	76	14	0	31	9	1.0	
(3446 LOC)	AT	read		195.00	0	0	0	265	195	195.0	
		write		195.00	0	0	0	121	195	195.0	
yacr2	FS	read	360	1.02	353	7	0	0	2	1.0	
		write	116	1.10	105	10	1	0	3	1.0	
(59 Functions)	FI	read		1.02	353	7	0	0	2	1.0	
		write		1.10	105	10	1	0	3	1.0	
(3979 LOC)	AT	read		106.00	0	0	0	360	106	106.0	
		write		106.00	0	0	0	116	106	106.0	
simulator	FS	read	485	1.32	431	5	39	10	13	1.0	*
		write	229	1.59	179	11	29	10	13	1.0	*
(111 Functions)	FI	read		1.37	429	1	30	25	13	1.0	
		write		1.66	179	9	16	25	13	1.0	
(4639 LOC)	AT	read		239.00	0	0	0	485	239	239.0	
		write		239.00	0	0	0	229	239	239.0	
099.go	FS	read	8378	1.12	8332	12	3	31	100	1.0	
		write	1236	1.48	1195	9	9	23	100	1.0	
(373 Functions)	FI	read		1.12	8332	12	3	31	100	1.0	
		write		1.48	1195	9	9	23	100	1.0	
(29637 LOC)	AT	read		180.00	0	0	0	8378	180	180.0	
		write		180.00	0	0	0	1236	180	180.0	
Total	FS	read	13105	1.21	12301	333	253	210	100	—	
		write	2602	1.46	2295	133	84	90	100	—	
	FI	read		1.24	12178	377	256	294	100	—	
		write		1.51	2260	139	76	127	100	—	
	AT	read		168.86	0	0	0	13105	239	—	
		write		146.33	0	0	0	2602	239	—	

Table 11: Part 2 of precision measurements counting all uses of "$*$", "$->$", "[ ]". An entry with a "$*$" indicates an increase in precision for the flow-sensitive analysis.