A Schema for Interprocedural Modification Side-Effect Analysis With Pointer Aliasing[*]

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

[†] Siemens Corporate Research Inc
755 College Rd. East
Princeton, NJ 08540
{wlandi,raltucher}@scr.siemens.com

[‡] Department of Computer Science
Rutgers University
Hill Center, Busch Campus
New Brunswick, NJ 08903
{ryder,pstocks,xxzhang}@cs.rutgers.edu

August 18, 1999

Abstract

The first interprocedural modification side-effects analysis for C (MOD$_C$) that obtains better than worst-case precision on programs with general-purpose pointer usage is presented with empirical results.[1] The analysis consists of an algorithm schema corresponding to a family of MOD$_C$ algorithms with two independent phases: one for determining pointer-induced aliases and a subsequent one for propagating interprocedural side effects. These MOD$_C$ algorithms are parameterized by the aliasing method used. The empirical results compare the performance of two dissimilar MOD$_C$ algorithms: MOD$_C$($FSAlias$) uses a flow-sensitive, calling-context-sensitive interprocedural alias analysis [LR92]; MOD$_C$($FIAlias$) uses a flow-insensitive, calling-context-insensitive alias analysis which is much faster, but less accurate. These two algorithms were profiled on 45 programs ranging in size from 250 to 30,000 lines of C code, and the results demonstrate dramatically the possible cost-precision tradeoffs. This *first comparative* implementation of MOD$_C$ analyses offers insight into the differences between *flow-/context-sensitive* and *flow-/context-insensitive* analyses. The analysis cost versus precision tradeoffs in side-effect information obtained is reported. The results show surprisingly that the precision of flow-sensitive side-effect analysis is not always prohibitive in cost, and that the precision of flow-insensitive analysis is substantially better than worst-case estimates and seems sufficient for certain applications. On average MOD$_C$($FSAlias$) for procedures and calls is in the range of 20% more precise than MOD$_C$($FIAlias$); however, the performance was found to be at least an order of magnitude slower than MOD$_C$($FIAlias$).

1 Introduction

Accurate compile-time calculation of possible interprocedural side effects is crucial for aggressive compiler optimization [ASU86], practical dependence analysis in programs with procedure

[*] The research reported here was supported, in part, by Siemens Corporate Research and NSF grants CISE-CCR-9208632, CCR-9501761, GER-9023628.

[1] This is a presentation of the algorithm schema for MOD$_C$ and describes new and extensive empirical results with two of the algorithms. The first MOD$_C$ algorithm in the schema was discussed in [LRZ93].

calls [Ban88, Pol88, Wol89], data-flow based testing [HS91, BH93, HFGO94, Ost90, Wey94, CR99, FW93, FI98], incremental semantic change analysis of software [Bur90, BR90, CR88, CK84, MR90a, MR91, PS89, Ryd83, RP88], interprocedural def-use relations [PLR94, HS94, GH98, CR99, Cha99] and effective static interprocedural program slicing [GL91, HRB90, OO84, RR95, GS96, LH96, HC98, SHR99, TCFR96, Tip96, Ven91, Wei84, TAFM97, AG96, AG98]. Many of these key applications in parallel and sequential programming environments need interprocedural def-use information which can be approximated using side-effect information. The utility of tools that address these problems can depend directly on the accuracy of the data-flow information available to them. Some problems may not need highly accurate data-flow information to solve them; in contrast, some applications may need to use all the information in a highly accurate solution. The latter applications need an efficient method to report program-point-specific side-effect information in the presence of pointers in order to handle modern languages such as C, C^{++}, FORTRAN90 and Java;[2] this requires practical interprocedural side-effect analysis with pointers, something that previous techniques for FORTRAN cannot supply [Ban79, Bur90, Coo85, CK88, CK87].

In the past, it has been suggested that one could do intraprocedural analyses of C codes, by using worst-case estimates of variables which could possibly experience a side effect at a call site. This yields a safe approximation of side-effect information, but almost surely overestimates the side effects in a program. To validate that program transformations preserve program semantics, however, more accurate side-effect information is needed.

This paper presents the first design and implementation of a *schema for practical interprocedural modification side effects* (i.e., MOD_C) for languages with general-purpose pointers (e.g., C). Since determination of pointer-induced aliasing occurs in the schema as a separable phase, the schema actually represents a family of MOD_C algorithms. The empirical experiments reported involve two MOD_C methods with different component aliasing algorithms. $\mathrm{MOD}_C(FSAlias)$ uses a flow-sensitive, calling-context-sensitive approximation algorithm for pointer-induced aliasing, called $FSAlias$ [LR92]; $\mathrm{MOD}_C(FIAlias)$ uses a flow-insensitive, calling-context-insensitive approximation algorithm for pointer-induced aliasing, called $FIAlias$, which is similar to the algorithm described in [ZRL96, Zha98]. The actual implemented algorithm handles unions and casting in C programs. The MOD_C schema is independent of the aliasing algorithm chosen and can use any aliasing algorithm, given a suitable interface. These MOD_C algorithms have extensive implementation results reported; these experiments are the first investigations of the cost-precision trade-offs of flow- and context-sensitivity as measured with respect to interprocedural side-effect analysis.

$\mathrm{MOD}_C(FSAlias)$ reports program-point-specific possible modification side effects; the results are more precise than information derivable using the same alias summary for all statements of a procedure. After aliases are computed, they are used to gather procedure summary modification information categorized by calling context, with subsequent propagation of modifications through the program call multigraph. Finally, call site modification information is calculated using the results of the procedure side-effects summary. $\mathrm{MOD}_C(FIAlias)$ also reports program-point-specific possible modification side effects, but it uses alias information that is assumed to be valid globally throughout the program. Thus, more spurious side effects may be reported locally and propagated on the call multigraph (in a context-insensitive manner).

The empirical tests of these algorithms used 45 C programs, most of which are publicly available.[3]

[2] Value-flow analysis for Java references is similar to pointer alias analysis.

[3] Visit **http://www.prolangs.rutgers.edu/** to obtain the public programs in this data set.

Measurements of average and maximum number of side effects found per assignment statement, per assignment through pointer dereference (i.e., a *through-dereference* assignment statement such as `*p=`), per procedure and per call have been recorded for both algorithms. Significantly, better precision is obtained by $\text{MOD}_C(FSAlias)$ at greater time cost than $\text{MOD}_C(FIAlias)$. This precision is necessary for some compiler transformations. $\text{MOD}_C(FSAlias)$ shows surprising scalability on programs up to 10,000 lines of code at compile-time cost in the prototype. Extensive use of recursive data structures is a key factor that limits the scalability of $\text{MOD}_C(FSAlias)$. Accordingly, $\text{MOD}_C(FSAlias)$ successfully analyses a 25,000 line program that doesn't use recursive data structures. Unexpectedly, $\text{MOD}_C(FIAlias)$ is much more accurate than a coarse worst-case estimate and costs at least an order of magnitude less than $\text{MOD}_C(FSAlias)$, so it may be sufficient and practical for program understanding applications on large codes. The decreased cost is primarily due to the cost of the $FIAlias$ phase relative to $FSAlias$. In addition, several suggested improvements may augment the precision of $\text{MOD}_C(FIAlias)$.

Specifically, the results for $\text{MOD}_C(FSAlias)$ show that procedures modify on average 11 locations, while $\text{MOD}_C(FIAlias)$ reports that procedures modify on average 17 locations, approximately 50% more side effects on average reported by the insensitive algorithm.[4] A crude measure of the accuracy of $\text{MOD}_C(FIAlias)$ versus $\text{MOD}_C(FSAlias)$ can be obtained by examining the difference in their solutions on the data admitting both kinds of analysis, since both are *safe* estimates of side effects that can occur. Normalized differences at calls and for procedures are presented and discussed in Section 4.

The empirical results show the utility of both analyses for specific applications and demonstrate the precision gains from sensitivity for certain data-flow information. Recent work in partitioning programs for analyses [ZRL98, ZRL96, Ruf97] yields hope that analyses of varying cost and precision can be applied to different parts of a program to obtain desired data-flow information at practical cost. The experiments reported here can be viewed as the initial investigation into the cost-precision tradeoffs involved when using data-flow analyses of varying degrees of flow- and context-sensitivity.

This paper is organized as follows. Section 2 discusses issues of accurate interprocedural data-flow analysis and pointer aliasing algorithms. Section 3 presents the MOD_C algorithm schema, its worst-case complexity and an example of both of the MOD_C algorithms used in the empirical tests. Section 4 reports the empirical results and derived observations. Section 5 details related work in data-flow analysis. Section 6 summarizes the contributions of the work. Appendix A presents a comparison of the MOD decomposition for C to that for FORTRAN. In Appendix B, an extended MOD algorithm based on our decomposition that approximately bounds the precision of side-effect solutions is explained. Finally, Appendix C presents the raw data from the empirical measurements discussed in Section 4.

2 Interprocedural Data-Flow Analysis

All interprocedural data-flow analyses for C-like languages encounter issues of problem formulation: how to obtain good static estimates of the possible execution paths through the program (including the possible calling patterns), how to treat variables created dynamically on the heap and aggregates

[4] If data for *moria*, a statistical outlier, is removed, the average total for $\text{MOD}_C(FIAlias)$ becomes 14. By contrast, removal of *moria* from the average total for $\text{MOD}_C(FSAlias)$ has no effect.

(i.e., arrays, structs), and how to obtain good approximations to the possible aliasing induced by pointer usage in the program.

2.1 Program Representation

A program is represented by a common directed graph structure, an ICFG or *interprocedural control flow graph*. This is no more than the control flow graph of each procedure connected together at call sites, each of which has been split into a *call* node and a *return* node. Procedures are made to have a single entry node and single exit node, if they do not already, by inserting extra (dummy) nodes and edges as required. Each call node is connected to the called procedure's entry node; each return node is connected to the called procedure's exit node. Figure 9 of Section 3.2 shows the ICFG representation of a small example program.

Iterative data-flow analysis is a fixed point calculation for recursive equations defined on a graph representing a program, that safely approximates the *meet over all paths solution* of a data-flow problem [Kil73, MR90b]. For interprocedural data-flow analysis, not all paths in the usual graph representation correspond to real program executions. A *realizable* path is a path on which every procedure returns to the call site which invoked it [JM82b, LR92, RHS95, SP81]. Paths on which a procedure does not return to the call site which invoked it are unrealizable and can never happen in an actual execution. (*setjump* and *longjump* are not allowed in the C programs analyzed.) A fundamental problem of interprocedural analysis is how to restrict the propagation of data-flow information to realizable paths, especially when the data-flow functions are *monotone* rather than *distributive* so that the fixed point solution need not be the meet-over-all-paths solution [KU77, MR90b].

2.2 Issues Involving Variables

The MOD_C schema defines a family of algorithms which determine modification side effects to fixed locations at program points. A *fixed location* is either an user-defined variable or a heap storage creation site. Each individual dynamically allocated fixed location is identified by the site that created it [JM82b, RM88]; therefore, whereas two fixed locations created at the same allocation site are not distinguishable, those created at different sites are. Fixed locations are so named because the relation between a fixed location and the storage location to which it refers, is unchanging during execution. Fixed locations do not contain any dereferences. For example in C syntax, `x` and `x.f` are fixed locations. By contrast, for other C names which include dereferences (e.g., `*p, p->f`), the relation between the name and the storage to which it refers can (and often does) change during execution. The term *object name* will be used to refer all C names in general, with or without dereferences; thus *fixed location* will delineate a subset of those names. Side effect information is obtained only for fixed locations.

All data-flow algorithms must deal with the *a priori* unbounded nature of recursive data structures. Many follow the approach of Jones and Muchnick [JM82a] which limits by truncation, the set of possible object names obtainable by following links in a recursive data structure, only maintaining the first k dereferences, a process known as *k-limiting*. Others have suggested less naive ways of restricting the namespace while obtaining more accurate aliases of heap-stored objects [CWZ90, Deu94, HPR89, LH88, HN90, HHN92, GH96a, GH96b, SRW98].

One difficulty with k-limiting is that it loses information about the suffix of an object name. With 2-limiting, the alias ⟨`p->f->g->h, x`⟩ is represented by ⟨`p->f->g#, x`⟩. However this k-limited object name also represents a set of object names, (e.g., `p->f->g->h`$_i$ for all i that make sense, `*(p->f->g)`, `p->f->g->h->j`). If no casting is allowed, then type information can rule out most erroneous cases. However, in the presence of casting, especially when an algorithm (like *FSAlias*) attempts to handle the most aggressive instances (e.g., casting a pointer value to an integer and back again to pointer type), an algorithm using this kind of k-limiting will experience much imprecision.

The *FSAlias* algorithm uses k-limiting of names in recursive data structures, based on Jones and Muchnick's original k-limiting definition, combined with a naming scheme that identifies a dynamically created fixed location by its creation site. *FIAlias* needs no k-limiting because it only reports aliases involving those object names that explicitly appear in a program.

As in most pointer-aliasing algorithms, arrays are treated as single variables by *FSAlias* and *FIAlias*. Some algorithms distinguish the independent fields of a structure (e.g., *FSAlias*, *FIAlias*, [EGH94, ZRL96, Ste96a]) while others do not (e.g., [SH97b, Ste96b]).

An added complication is presented by *non-visible* object names. The *non-visible*s are local variables of procedures live at the call site (or in an earlier invocation of the current procedure) which are accessible through an alias, although not visible directly in the current scope. Possible side effects to these object names must be accounted for [LR92, LRZ93, EGH94].

2.3 Alias Representation

The MOD$_C$ schema requires knowledge of *aliases,* that is, object names that may refer to the same storage location at some point in the execution. In programming languages such as C, explicit addressing operators render alias analysis more difficult than in FORTRAN, where aliases are introduced only through call-by-reference parameter passing. But the need for alias analysis still exists in modern programming languages whose pointer usage is more constrained (e.g., Java and FORTRAN90).[5]

Alias algorithms can be distinguished by their representation of the alias relations and the degree to which they preserve program-point-specific information. Aliases are either represented explicitly as pairs of object names or implicitly embedded in a points-to relation.[6] Hendren *et al.* represent aliases as a set of simultaneous points-to relations at a particular program point [HN90, EGH94] (e.g.,⟨`x,y`⟩) means `x` points to the object name `y`). Choi *et al.* use an *implicit* representation which stores all aliases as pairs consisting of a fixed location and the object name (containing a single dereference) which points to it (e.g., ⟨∗`p,x`⟩ means the dereferenced value of `p` is `x`). This representation, like a points-to representation, requires a closure step to obtain object names containing multiple levels of dereferences [CBC93, MLR$^+$93, BCCH94, BCCH97]. For example, ⟨∗`p`,∗`q`⟩ and ⟨∗`q,r`⟩ must be combined to yield ⟨∗∗`p,r`⟩. *FSAlias* [LR92], the flow-/context-sensitive alias approximation algorithm uses an *explicit* representation of aliases as pairs of object names possibly containing dereferences (e.g., ⟨∗∗`p`,∗`q`⟩). Redundant aliases obtained through dereferences applied

[5] The MOD$_C$ schema would be largely unchanged for these languages although the alias phase would probably be more efficient by being specialized to their simpler pointer usages.

[6] Differences between *points-to* analysis notation and the explicit pointer alias representation were also discussed in [EGH94].

to both elements of an alias pair are not stored explicitly but are inferred (e.g., $\langle *\texttt{p},*\texttt{q}\rangle$ implies $\langle **\texttt{p},**\texttt{q}\rangle$). These two representations will be referred to by the terms *explicit* and *implicit*.

Essentially, the implicit representation is the same as that in the points-to relation, except the points-to has the ”*” implied for the first element, making the order of the relation relevant. By explicitly requiring the ”*” to appear, the implicit representation can also directly represent reference parameter and structural aliases between object names, but not general aliasing (e.g., $\langle **\texttt{p},*\texttt{q}\rangle$); all of these are directly representable with the explicit representation.

The implicit representation is claimed to be a space savings over the explicit representation, but no empirical or theoretical comparison has yet been made. The observed memory needs of the explicit representation used in $FSAlias$, are bounded by the memory measurements in Section 4. While it is true that the implicit representation is more compact,[7] the explicit representation allows for optimizations that interprocedurally enable the alias analysis only to pass a part of the solution from the caller's context into the called procedure, which can result in significant space savings, especially on larger programs. Thus, the memory tradeoffs in choice of representation are not clear.

The implicit and explicit representations can be shown incomparable in terms of the resulting accuracy they exhibit as illustrated in Figure 1. In Figure 1(a), an algorithm using the implicit representation will result in the $\langle *\texttt{x},\texttt{z}\rangle$ alias replacing incoming alias $\langle *\texttt{x},\texttt{y}\rangle$ at statement `S3`; thus no aliases involving `y` will be reported at `S4`. By contrast, an algorithm using the explicit representation will have formed the alias $\langle **\texttt{p},\texttt{y}\rangle$ at statement `S2` and it will remain in the alias set (as a spurious alias) reported at statement `S4`. In Figure 1(b), an algorithm using the implicit representation will combine the aliases $\langle *\texttt{p},\texttt{q}\rangle$ and $\langle *\texttt{q},\texttt{y}\rangle$ from opposite arms of the if statement and obtain the spurious alias $\langle **\texttt{p},\texttt{y}\rangle$ at `S5`, since this representation implies combination of any alias pairs at a program point when aliases are needed. An algorithm using the explicit representation will not find this spurious alias because no transitive combination of alias pairs is required. In Figure 1(c), an algorithm using either the implicit or explicit representation will wrongly combine incoming aliases at statement `S5` $\langle *\texttt{p},\texttt{q}\rangle$ and $\langle *\texttt{r},\texttt{s}\rangle$ with the unconditional alias created at that statement, $\langle *\texttt{q},\texttt{r}\rangle$, obtaining the spurious alias $\langle ***\texttt{p},\texttt{s}\rangle$. This occurs because neither representation encodes enough information to remember that the two incoming aliases do not exist concurrently on a path in the program. These examples are simple instances of the general problem.

```
S1:  x = &y;
S2:  p = &x;
S3:  x = &z;
S4:  **p =
```

(a) Implicit representation more precise

```
S1:  q = &z;
S2:  if ()
S3:     p = &q;
     else
S4:     q = &y;
S5:  **p =
```

(b) Explicit representation more precise

```
S1:  if ()
S2:     p = &q;
     else
S3:     r = &s;
S4:  q = &r;
S5:  ***p =
```

(c) Both imprecise

Figure 1: Comparison of implicit and explicit representations of aliases.

[7] The implicit representation is called *compact* in [HP98].

2.4 Analysis Precision and Sensitivity

Since the basic problem of determining pointer-induced aliases is intractable for programs with multiple levels of indirection [LR91, Lan92b, Ram94], practical pointer-aliasing algorithms are approximate. Many algorithms use intraprocedural propagation of aliases through pointer-assignment statements in a manner conceptually similar to the single-level pointer aliasing algorithm in Chapter 10 of [ASU86] with extensions to handle multiple-level pointers. Intraprocedural algorithms can make worst-case assumptions about the effects of call sites and determine a "first-cut" alias solution.

Most pointer aliasing algorithms now in the literature do some form of interprocedural analysis. They are distinguishable by the amount and type of calling context they preserve with the derived alias information. Some algorithms obtain differentiated program-point-specific alias information, because in these algorithms statement order of execution is significant. They are called *flow-sensitive* methods. When flow of control of execution is ignored, a *flow-insensitive* method is obtained. Algorithms which propagate alias information across calls, along paths in the called procedure and then back again into the calling procedure, keeping approximate calling-context information with each alias pair, are termed *context-sensitive*, in that they distinguish back propagation of information between different call sites. *Program-wide* alias information is obtained by techniques which upon identifying an alias, presume it holds throughout the program. This can be done in a context-sensitive or insensitive manner (e.g., by replicating fixed-location names).

2.5 Maintaining Calling Context

There are many methods proposed for distinguishing calling contexts (i.e., the state of the call stack) in data-flow algorithms. Sharir and Pnueli [SP81] advocate the use of a call-string list of open and not yet closed procedure activations to label data-flow information precisely with the calling context in which it was obtained. They also suggest use of an approximate call-string consisting of the last j calls on the call stack. The call-string list is close to the approach used in the points-to algorithm developed at McGill University, where every procedure activation is analyzed separately [HN90, EGH94]; optimizations to reduce computation by reusing the results for similar calling contexts were suggested by Emami [Ema93], and have been subsequently developed by Wilson and Lam [WL95] for points-to analysis and Ghiya and Hendren for connection analysis [GH96a]. Empirical data seems to suggest that such optimizations can dramatically reduce the number of contexts actually analyzed. Jones and Muchnick [JM82b] describe the use of an abstraction of the calling context at a dynamic creation site for a variable; the precision of this abstraction plus the approximation lattice for the data-flow problem in question determine the precision of the solution. Choi *et al.* use the immediate past call site as their encoding of the calling context in their flow-sensitive aliasing algorithm [CBC93, MLR$^+$93, BCCH97]. They also describe an algorithm variant that uses alias sets of unrestricted size at the call site, called *source alias sets*, as additional call site encoding information. Their use of the previous call site name is the same as approximations suggested earlier by [JM82b, SP81]. It also resonates with later work in the functional programming community on higher-order functions, Shivers' control flow analyses (CFA) [Shi88] in which a suffix of the call stack contents is used to approximate calling context (e.g., 0CFA - no call sites distinguished, 1CFA - last call site distinguished, etc.).

The calling context approximation used in the MOD$_C$ schema is inherited from the alias analysis used. For $\text{MOD}_C(\mathit{FSAlias})$, this is the same as that of the *FSAlias* algorithm [LR91, LR92]. The data-flow fact that x and y are aliased at program point n is represented by an unordered pair $\langle$x,y$\rangle$ at n. The encoding of calling context is the set of *reaching aliases*[8] (*RA*s) that exists at entry of procedure p containing n when p is invoked from a particular call site. When an alias exists independently of calling context, any reaching alias is an appropriate context to use, but for convenience the special reaching alias ϕ is used. The *RA* set can be used to determine to which call sites, aliases at the exit of a called procedure should be propagated, namely only to those call sites which induce that *RA* set. Essentially the *RA* set induced by a call corresponds to a source alias set to which a namespace mapping is applied that includes the parameter bindings as well as scoping transformations. Using a single alias pair from the *RA* set to determine calling context, yields a safe approximate solution of realizable paths for programs containing multiple levels of dereferencing; this is the approximation used for calling context in $\text{MOD}_C(\mathit{FSAlias})$ at procedure entry. For aliasing in programs restricted to one level of dereferencing, the *RA* sets are of cardinality one and can be used to obtain a precise solution [LR91]. The empirical results in Section 4 indicate that this is also a good approximation in practice. *RA* is used in the description of the MOD_C schema to represent some approximation of calling context.

Figures 2 and 3 show that the reaching alias encoding of calling context is incomparable to using the last call site (i.e.,1CFA). For comparison purposes, these examples have been coded using the Landi-Ryder representation of aliases. In these examples aliases created independently of calling context are labeled with call site $\perp$ and spurious aliases are underlined. In Figure 2, the approximation arises because before n_3, an algorithm using reaching aliases for calling context cannot determine that $\langle *$p,q$\rangle$ and $\langle *$x,y$\rangle$ never occur on the same path; since different reaching aliases might correspond to the same call site, the safe approximation is to assume this does occur, which causes a spurious alias to be created. By using last call site information, however, an algorithm can see that each alias is labeled by a different call site.

In Figure 3, the approximation in using last call site arises because on the return of B to A, the algorithm has lost all information differentiating the call sites in the main program, whereas in this case, the reaching aliases distinguish the call sites; unlike in Figure 2, there is no approximation since no aliases are created involving their possible interaction. Thus, these two calling context approximations are incomparable.

2.6 The *FSAlias* and *FIAlias* Algorithms

The MOD_C schema inherits its calling-context sensitivity and flow sensitivity from the pointer-aliasing algorithm used. The empirical tests have exercised two specific choices of MOD_C algorithms at opposite ends of the sensitivity spectrum, namely $\text{MOD}_C(\mathit{FSAlias})$ which is flow-/context-sensitive, and $\text{MOD}_C(\mathit{FIAlias})$ which is flow-/context-insensitive. $\text{MOD}_C(\mathit{FSAlias})$ is more costly and more accurate, in general, than $\text{MOD}_C(\mathit{FIAlias})$ because of the differences in cost and accuracy of the aliasing algorithms used. These techniques allow exploration of the precision/cost tradeoffs of side-effects analysis and the scalability of these approaches applied to real programs. While the results obtained are specific to the two algorithms used, it seems likely that similar results will be obtainable from any pair of algorithms which vary similarly in sensitivity. More general claims would require additional experimentation.

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

	reaching alias	last call site
`int **p, *q, *r;`		
`int *x, y;`		
`void main ()`		
`{`		
`p = &r;`	$\{\ [\phi, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$	$\{\ [\perp, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$
`if () {`		
`p = &q;`	$\{\ [\phi, \langle *\mathtt{p}, \mathtt{q}\rangle]\ \}$	$\{\ [\perp, \langle *\mathtt{p}, \mathtt{q}\rangle]\ \}$
n_1 : `A ();`	$\left\{\begin{array}{l} [\phi, \langle *\mathtt{p}, \mathtt{q}\rangle],\ \ [\phi, \langle **\mathtt{p}, *\mathtt{x}\rangle], \\ [\phi, \langle *\mathtt{q}, *\mathtt{x}\rangle],\ \ \underline{[\phi, \langle *\mathtt{q}, \mathtt{y}\rangle]} \end{array}\right\}$	$\left\{\begin{array}{l} [\perp, \langle *\mathtt{p}, \mathtt{q}\rangle],\ \ [\perp, \langle **\mathtt{p}, *\mathtt{x}\rangle], \\ [\perp, \langle *\mathtt{q}, *\mathtt{x}\rangle] \end{array}\right\}$
`} else {`		
`x = &y;`	$\{\ [\phi, \langle *\mathtt{x}, \mathtt{y}\rangle], [\phi, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$	$\{\ [\perp, \langle *\mathtt{x}, \mathtt{y}\rangle], [\perp, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$
n_2 : `A ();`	$\left\{\begin{array}{l} [\phi, \langle *\mathtt{x}, \mathtt{y}\rangle],\ \ [\phi, \langle **\mathtt{p}, *\mathtt{x}\rangle], \\ [\phi, \langle **\mathtt{p}, \mathtt{y}\rangle],\ \ [\phi, \langle *\mathtt{r}, *\mathtt{x}\rangle], \\ [\phi, \langle *\mathtt{r}, \mathtt{y}\rangle],\ \ [\phi, \langle *\mathtt{p}, \mathtt{r}\rangle] \end{array}\right\}$	$\left\{\begin{array}{l} [\perp, \langle *\mathtt{x}, \mathtt{y}\rangle],\ \ [\perp, \langle **\mathtt{p}, *\mathtt{x}\rangle], \\ [\perp, \langle **\mathtt{p}, \mathtt{y}\rangle],\ \ [\perp, \langle *\mathtt{p}, \mathtt{r}\rangle] \\ [\perp, \langle *\mathtt{r}, *\mathtt{x}\rangle],\ \ [\perp, \langle *\mathtt{r}, \mathtt{y}\rangle] \end{array}\right\}$
`}`		
`}`		
`void A ()`		
`{`	$\left\{\begin{array}{l} [\langle *\mathtt{p}, \mathtt{q}\rangle, \langle *\mathtt{p}, \mathtt{q}\rangle],\ \ [\langle *\mathtt{x}, \mathtt{y}\rangle, \langle *\mathtt{x}, \mathtt{y}\rangle], \\ [\langle *\mathtt{p}, \mathtt{r}\rangle, \langle *\mathtt{p}, \mathtt{r}\rangle] \end{array}\right\}$	$\left\{\begin{array}{l} [n_1, \langle *\mathtt{p}, \mathtt{q}\rangle],\ \ [n_2, \langle *\mathtt{x}, \mathtt{y}\rangle], \\ [n_2, \langle *\mathtt{p}, \mathtt{r}\rangle] \end{array}\right\}$
n_3 : `*p = x;`		
`}`	$\left\{\begin{array}{l} [\langle *\mathtt{p}, \mathtt{q}\rangle, \langle *\mathtt{p}, \mathtt{q}\rangle],\ \ [\langle *\mathtt{x}, \mathtt{y}\rangle, \langle *\mathtt{x}, \mathtt{y}\rangle], \\ [\phi, \langle **\mathtt{p}, *\mathtt{x}\rangle],\ \ [\langle *\mathtt{p}, \mathtt{q}\rangle, \langle *\mathtt{q}, *\mathtt{x}\rangle], \\ [\langle *\mathtt{x}, \mathtt{y}\rangle, \langle **\mathtt{p}, \mathtt{y}\rangle],\ \ [\langle *\mathtt{p}, \mathtt{r}\rangle, \langle *\mathtt{r}, *\mathtt{x}\rangle], \\ [\langle *\mathtt{p}, \mathtt{r}\rangle, \langle *\mathtt{r}, \mathtt{y}\rangle],\ \ [\langle *\mathtt{p}, \mathtt{r}\rangle, \langle *\mathtt{p}, \mathtt{r}\rangle], \\ \underline{[\langle *\mathtt{p}, \mathtt{q}\rangle, \langle *\mathtt{q}, \mathtt{y}\rangle]} \end{array}\right\}$	$\left\{\begin{array}{l} [n_1, \langle *\mathtt{p}, \mathtt{q}\rangle],\ \ [n_2, \langle *\mathtt{x}, \mathtt{y}\rangle], \\ [n_2, \langle *\mathtt{p}, \mathtt{r}\rangle],\ \ [\perp, \langle **\mathtt{p}, *\mathtt{x}\rangle], \\ [n_1, \langle *\mathtt{q}, *\mathtt{x}\rangle],\ \ [n_2, \langle **\mathtt{p}, \mathtt{y}\rangle], \\ [n_2, \langle *\mathtt{r}, *\mathtt{x}\rangle],\ \ [n_2, \langle *\mathtt{r}, \mathtt{y}\rangle] \end{array}\right\}$

Figure 2: An example in which last call site is more precise than reaching alias.

		reaching alias	last call site
	`int *p, q, r;`		
	`void main ()`		
	`{`		
	`p = &q;`	$\{\ [\phi, \langle *\mathtt{p}, \mathtt{q}\rangle]\ \}$	$\{\ [\perp, \langle *\mathtt{p}, \mathtt{q}\rangle]\ \}$
n_1 :	`A ();`	$\{\ [\phi, \langle *\mathtt{p}, \mathtt{q}\rangle]\ \}$	$\{\ [\perp, \langle *\mathtt{p}, \mathtt{q}\rangle],\ \underline{[\perp, \langle *\mathtt{p}, \mathtt{r}\rangle]}\ \}$
	`p = &r;`	$\{\ [\phi, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$	$\{\ [\perp, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$
n_2 :	`A ();`	$\{\ [\phi, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$	$\{\ \underline{[\perp, \langle *\mathtt{p}, \mathtt{q}\rangle]},\ [\perp, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$
	`}`		
	`void A ()`	$\{\ [\langle *\mathtt{p}, \mathtt{q}\rangle, \langle *\mathtt{p}, \mathtt{q}\rangle],\ [\langle *\mathtt{p}, \mathtt{r}\rangle, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$	$\{\ [n_1, \langle *\mathtt{p}, \mathtt{q}\rangle],\ [n_2, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$
	`{`		
n_3 :	`B ();`		
	`}`	$\{\ [\langle *\mathtt{p}, \mathtt{q}\rangle, \langle *\mathtt{p}, \mathtt{q}\rangle],\ [\langle *\mathtt{p}, \mathtt{r}\rangle, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$	$\{\ [\perp, \langle *\mathtt{p}, \mathtt{q}\rangle],\ [\perp, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$
	`void B ()`	$\{\ [\langle *\mathtt{p}, \mathtt{q}\rangle, \langle *\mathtt{p}, \mathtt{q}\rangle],\ [\langle *\mathtt{p}, \mathtt{r}\rangle, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$	$\{\ [n_3, \langle *\mathtt{p}, \mathtt{q}\rangle],\ [n_3, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$
	`{`		
	`}`	$\{\ [\langle *\mathtt{p}, \mathtt{q}\rangle, \langle *\mathtt{p}, \mathtt{q}\rangle],\ [\langle *\mathtt{p}, \mathtt{r}\rangle, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$	$\{\ [n_3, \langle *\mathtt{p}, \mathtt{q}\rangle],\ [n_3, \langle *\mathtt{p}, \mathtt{r}\rangle]\ \}$

Figure 3: An example in which reaching alias is more precise than last call site.

The following alias algorithm descriptions are explained on an intuitive level to stress key concepts, with an example following to show the differences between *FSAlias* and *FIAlias*. Detailed comprehension of these algorithms is not necessary for understanding this paper.

2.6.1 *FSAlias*

Figure 4 gives an overview of the *FSAlias* algorithm, which relies on a flow-/context-sensitive fixed point iteration on the ICFG, using a standard worklist approach. In *FSAlias*, alias information is propagated along the static paths in each procedure for a specific calling context (i.e., a reaching alias) in a manner which preserves statement order; during this propagation, aliases are created or destroyed depending on the semantics of the program statements encountered on the path. Therefore, a full description of the algorithm requires a description of the transfer functions at intraprocedural nodes (i.e., pointer assignments) and at interprocedural nodes (i.e., call sites).

The initialization phase of the algorithm (step 1) populates the initial worklist with the initial set of aliases, either those created intraprocedurally by a pointer assignment (1.1) or interprocedurally by parameter-argument associations at calls (1.2). This is done by procedures *alias_intro_by_assignment* and *aliases_intro_by_call* respectively. Aliases that are created regardless of any reaching alias can legitimately be associated with any reaching alias, but for practicality are only associated with a special reaching alias, ϕ.

Step 2 performs the data-flow information propagation both intraprocedurally and interprocedurally. During one step in the iteration, a (*node,* [*calling context, alias*]) entry is removed from the worklist, and propagated to successor nodes by invoking appropriate handling procedures. (n, [*RA, PA*]) represents the fact that alias PA holds at node n in calling context RA. The term *alias tuple* will be used to refer to an alias pair with its corresponding calling context (e.g., [*RA, PA*]).

The intraprocedural propagation of aliases through pointer assignment statements is described by transfer functions associated with each node in a standard extension of [ASU86] (i.e., performed by procedure *alias_implies_thru_assign* in step 2.4.1). If an intraprocedural node has no effect on pointer aliasing then the tuple is preserved through that node (i.e., performed by procedure *preserve* in step 2.4.2).

For example, in Figure 7 at the exit of statement 8 (`p = q`) in `main`, the alias tuple $[\phi,\langle *\mathtt{p}, *\mathtt{q}\rangle]$ is generated under all calling contexts. The alias tuple $[\phi,\langle *\mathtt{q}, \mathtt{a}\rangle]$ reaches (and is preserved through) statement 8 after having been created at statement 7. This alias is combined with the semantics of the pointer assignment at 8 to create alias $[\phi,\langle *\mathtt{p}, \mathtt{a}\rangle]$ at the exit of statement 8. This fully describes the transfer function associated with the pointer assignment at statement 8. Similar functions are used at all pointer assignments [LR92]. Essentially, an alias is created regardless of calling context by the assignment itself, current aliases of the left-hand-side object name (assuming it can be dereferenced) are killed by this assignment, and aliases of the right-hand-side object name (if any) become new aliases of the dereferenced left-hand-side object name.

Intuitively, the processing in step 2 for propagation of aliases across procedure boundaries occurs as follows. Interprocedurally, a call to procedure Q, $call_Q$, creates reaching aliases at the entry of Q. If the algorithm is analyzing the calling procedure under calling context RA, $contexts\_of(call_Q, RA)$ denotes the set of reaching aliases induced by both the parameter bindings (handled by step 1.2) *and* the aliases associated with RAs which reach the call. The special reaching alias ϕ and reaching aliases created solely by the parameter bindings are included in the set $contexts\_of(call_Q, \phi)$.[9] $Calias(n, RA)$ represents the set of aliases at program point n under the calling context RA which reaches the entry of the procedure containing n.[10] The mapping of $Calias(n, RA)$ into the called procedure is handled by procedure *alias_at_call_implies* in step 2.2.

At the exit of Q, aliases associated with reaching alias RA' are propagated to any call site $call_Q$, where $RA' \in contexts\_of(call_Q, RA)$, and thereafter are associated with RA in the procedure containing that call site. This mapping is performed by procedure *alias_at_exit_implies* in step 2.3. Aliases associated with reaching alias ϕ are valid at every call site. The details of the algorithm include the namespace mappings between the calling and called procedures [LR92].

For example, in Figure 7 the set of alias tuples which reach the call at statement 10 is $([\phi,\langle *\mathtt{p}, *\mathtt{q}\rangle],\ [\phi,\langle *\mathtt{q}, \mathtt{a}\rangle],\ [\phi,\langle *\mathtt{p}, \mathtt{a}\rangle])$. The transfer function at the call maps each of these aliases into `proc1` as a reaching alias, so that the alias tuples at exit of statement 3 in `proc1` are: $([\langle *\mathtt{p,*q}\rangle,\langle *\mathtt{p}, *\mathtt{q}\rangle],\ [\langle *\mathtt{q,a}\rangle,\langle *\mathtt{q}, \mathtt{a}\rangle],\ [\langle *\mathtt{p,a}\rangle,\langle *\mathtt{p}, \mathtt{a}\rangle])$. At the exit of statement 4 in `proc1`, we preserve all of these incoming alias tuples and generate new alias tuples: $([\phi,\langle *\mathtt{r}, *\mathtt{q}\rangle],\ [\langle *\mathtt{q,a}\rangle,\langle *\mathtt{r}, \mathtt{a}\rangle])$. Now, at the return from the call at statement 10 (recall that the ICFG breaks all call statements into two nodes, a *call* and a *return*), step 2.3 of the algorithm in Figure 4 propagates the alias tuples $([\phi,\langle *\mathtt{p}, *\mathtt{q}\rangle],\ [\phi,\langle *\mathtt{q}, \mathtt{a}\rangle],\ [\phi,\langle *\mathtt{p}, \mathtt{a}\rangle],\ [\phi,\langle *\mathtt{r}, *\mathtt{q}\rangle],\ [\phi,\langle *\mathtt{r}, \mathtt{a}\rangle])$, back to the return at statement 10. Note that with respect to the last tuple, RA is ϕ in the previous discussion and RA' is $\langle *\mathtt{q,a}\rangle$. This examplifies the mapping of corresponding reaching aliases at call sites during alias propagation.

[9] The correspondence between the non-visibles and non-addressable object names is memoized at the called procedure entry so that a local of a (possibly transitive) calling procedure is mapped to the representative object name *nv* which is used throughout the alias propagation on paths from that procedure entry. Upon return from a call, any aliases involving *nv* are expanded using the memoized information into the actual alias relations at the appropriate call sites.

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

```
begin
    1. for each node $n$ in $ICFG$
        1.1  if $n$ is a pointer assignment
                 $aliases\_intro\_by\_assignment(n)$;
        1.2  else if $n$ is a call node
                 $aliases\_intro\_by\_call(n)$;
    2. while $worklist$ is not empty
        2.1  remove $(n, [RA, PA])$ from $worklist$;
        /* interprocedural propagation */
        2.2  if $n$ is a call node
                 $alias\_at\_call\_implies(n, RA, PA)$;
        2.3  else if $n$ is an exit node
                 $alias\_at\_exit\_implies(n, RA, PA)$;
        2.4  else /* intraprocedural propagation */
                 for each $m \in successor(n)$
                 2.4.1  if $m$ is a pointer assignment
                            $alias\_implies\_thru\_assign(m, RA, PA)$;
                 2.4.2  else
                            $preserve(m, RA, PA)$;
end
```

Figure 4: *FSAlias* algorithm

The main approximation which can occur in *FSAlias* is that sometimes two incoming aliases at a pointer assignment that are needed to create an alias as a side effect of that statement, can never actually occur on the same execution path from program start to this assignment. There is an example of this in Figure 1(c). At the exit of statement `S4`, the existence of aliases ⟨∗`p,q`⟩ and ⟨∗`r,s`⟩ combined with the assignment at `S4` seem to imply that ⟨∗∗∗`p,s`⟩, but there is no execution path through the if statement on which both ⟨∗`p,q`⟩ and ⟨∗`r,s`⟩ hold. The assumption about incoming alias information is that sets of aliases can hold simultaneously on some path to this program point; this assumption may lead to safe but possibly imprecise aliases being created.

FSAlias has worst-case polynomial time complexity.

2.6.2 *FIAlias*

FIAlias is a fast, coarse-grained alternative to *FSAlias*, and is similar to various other flow-/context-insensitive algorithms [Cou86, Gua88, BCCH94, And94, SH97b, Ste96b, ZRL96, Zha98, Wei80, HP98]. Aliasing is expressed as a relation between pairs of object names, which is symmetric and reflexive, but not transitive; however, it is a common (though not universal) practice to approximate this relation with transitive solutions. *FIAlias* forms a partition of the object names which is a transitive representation of aliasing, and clearly can be represented with space linear in the number of object names in the program.

For simplicity of explication, assume (i) there are no structure assignments (These can be broken into multiple non-structure assignments.), (ii) all functions are of type `void` (There are many ways to handle returned values; for example, a function can be made to assign its return value to some new global and that global can be used to retrieve the returned value at the call site.), and (iii) there is no casting and no union types. (The handling of casting and union types is simply a matter of encoding the relationship of fields within object names [YHR99].) Either fields can be represented by offsets from the start of the structure [WL95, EGH94] or by symbolic names [Ste96b, ZRL96]. *FIAlias* uses the first approach.

The basic idea of *FIAlias* is that for the assignment `a = b`, ∗`a` and ∗`b` become aliased (this is called a *type 1 alias effect*) and so do $*^i$`a` and $*^i$`b` for all $i \geq 2$, assuming those object names make sense (this is called a *type 2 alias effect*). The type 2 alias effect induces the right-regular property discussed in [Deu92, ZRL96].

The *FIAlias* algorithm partitions the program's object names using a union/find data structure [ASU86]. An overview of the steps of *FIAlias* is:

1. Consider each object name to be in its own partition element.

2. Perform unions of partition elements to account for type 1 alias effects at all assignments, with formal/actual bindings at call sites considered to be assignments.

3. Perform unions of partition elements to account for type 2 alias effects.

4. Perform some additional unions to ensure that actuals for function pointer call sites are unioned with the formals of the called procedure. When handling functions that are non-`void`, the returned value must be unioned with the name to which the value is assigned.

Typically the only alias information of interest concerns object names that physically appear in the program and their fixed-location aliases. The set of *interesting* object names is defined to be those object names needed to get an explicit alias solution for all object names that appear in the program and all fixed locations (i.e., including heap storage creation site names). If an object name `n` appears in the program, then `n`, ∗`n`, and any prefix of `n` are interesting; thus, if `p->m` appears in the program, then ∗`(p->m)`, `p->m` and the prefixes ∗`p` and `p` are all interesting. Extending the interesting object names by one dereference is necessary to compute the aliases of ∗`p` given `p=q`, since the aliases of ∗`q` must be known.

The *FIAlias* algorithm is presented in Figure 5. In Phase 1, an object name `name` is *plausible* if there is any possibility that ∗`name` can have aliases; thus, constants are not plausible object names, and assignments like `i=5` and `p=NULL` generate no plausible left-hand-side or right-hand-side pairs. An assignment `i=j` where `i` and `j` are integers is a more complicated case; this statement can generate the plausible pair `i/j` if pointer-valued data is assumed to be transferable through integer object names using casts, as in the C program in Figure 6. Similarly, for `p=q+i` where `p` and `q` are pointers and `i` is an integer, `p/q` must be considered as a plausible left-hand-side/right-hand-side pair, but `p/i` may not be considered as such. The assignment `function_pointer = function` is treated as `function_pointer = &function` for this analysis; that is, ∗`function_pointer` must be unioned to `function` and *not* to ∗`function`.

Phase 2 simply makes sure that for all pairs $\texttt{name}_i$, $\texttt{name}_j$ in a partition, that $*\texttt{name}_i$ and $*\texttt{name}_j$ are also in the same partition (i.e., type 2 alias effects). Phase 2 defines `map` so that for all object names `n`, map[find(`n`)][`op`] = find(`op(n)`) assuming `op(n)` is interesting. For example, map[find(`n`)][∗] = find(∗`n`) if ∗`n` is interesting. The type signature of `map` is

```
begin
/* Phase 1:  Creating initial partitions */

    for each plausible left-hand-side (lhs), right-hand-side (rhs) pair in the program
        union(find(*lhs),find(*rhs))

/* Phase 2:  Refining partitions for Type 2 alias effects */

    for each partition part
        map[part] = $\lambda x$.NULL (everything maps to NULL)
    for each interesting name X which is an operation op applied to a name Y
    /* For example:  X=*p, Y = p, and op = *;      */

    /* The outermost find is needed because a partition */
    /* is represented by a representative element.  Since union can change */
    /* which element is the representative element, a find */
    /* is needed here.  For simplicity, find(NULL) is NULL. */
        if find(X) $\neq$ find(map[find(Y)][op])
            then
                if map[find(Y)][op] is NULL
                    then redefine map[find(Y)][op] to be find(X)
                    else merge(find(X),find(map[find(Y)][op]))

/* Phase 3:  Accounting for calls through function pointers */

    repeat
        for each call through a function pointer which has arguments (fp($\text{arg}_1$,...,$\text{arg}_n$))
            for each function func in the partition of fp
                merge(find(*$\text{arg}_i$),find(*$\text{form}_i$)) for all $1 \leq i \leq n$
                where $\text{form}_i$ is the $i^{th}$ formal of func
    until no changes in the partitions involving function pointers.

end
/* merge function used in Phases 2,3 */

    function merge($\text{part}_1$,$\text{part}_2$)
    begin   if find($\text{part}_1$) $\neq$ find($\text{part}_2$)
            let $\text{old}_1$ = map[find($\text{part}_1$)] and $\text{old}_2$ = map[find($\text{part}_2$)]
            union($\text{part}_1$,$\text{part}_2$)
            map[find($\text{part}_1$)] = $\lambda x. \left( \begin{array}{ll} \text{NULL} & \text{if } \text{old}_1[x] = \text{old}_2[x] = \text{NULL} \\ \text{old}_1[x] & \text{if } \text{old}_2[x] = \text{NULL} \\ \text{old}_2[x] & \text{if } \text{old}_1[x] = \text{NULL} \\ \textit{merge}(\text{old}_1[x],\text{old}_2[x]) & \text{otherwise} \end{array} \right)$
        return find($\text{part}_1$)
    end
```

Figure 5: *FIAlias* algorithm

```
int *p, i, j, k;
j = (int)&k;
i = j;
p = (int *)i;
```

Figure 6: Unsafe use of casting

$$\mathtt{map} : \mathit{partition} \rightarrow\ (\mathit{operation} \rightarrow \mathit{partition}).$$

Map is built incrementally by iterating through all interesting object names. The merge function, also presented in Figure 5, unions two partitions and updates map.

Phase 3 is straightforward, accounting for calls through function pointers. fp can be an arbitrary object name; for example, $\mathtt{var}_1$, $*\mathtt{var}_1$, or $**\mathtt{var}_2$. Care must be taken in C because the calls var(...) and (*var)() are semantically equivalent and should be be treated as such.

2.6.3 Example.

Figure 7 demonstrates the differences in precision between *FSAlias* and *FIAlias* through a small example involving two procedures, main and proc1. First, consider the solution obtained by *FIAlias*. Since q is assigned both the address of a and b in main, *FIAlias* will assume that the aliases $\langle *\mathtt{q},\mathtt{a}\rangle$ and $\langle *\mathtt{q},\mathtt{b}\rangle$ hold globally throughout the program. Because *FSAlias* propagates alias information through statements 7-14 in order, the alias $\langle *\mathtt{q},\mathtt{a}\rangle$ holds on exit from statement 7, through statements 8-9, and is killed by statement 11; therefore, *FSAlias* can tell that b is the only fixed location aliased to $*\mathtt{q}$ at statement 12, not both a and b which would be reported by *FIAlias*. Similarly, at statement 9, *FSAlias* can tell that only fixed location a is aliased to $*\mathtt{p}$ whereas *FIAlias* cannot and reports $*\mathtt{p}$ aliased to both a and b.

To illustrate the interprocedural propagation of aliases, proc1 is called by main at statements 10 and 14. The alias tuple $[\phi,\langle *\mathtt{q},\mathtt{a}\rangle]$ reaches the exit of statement 9. This is propagated into proc1 as alias tuple $[\langle *\mathtt{q},\mathtt{a}\rangle,\langle *\mathtt{q},\mathtt{a}\rangle]$ and reaches the exit of statement 3. Then, alias tuple $[\langle *\mathtt{q},\mathtt{a}\rangle,\langle *\mathtt{r},\mathtt{a}\rangle]$ is created on exit of statement 4 and it and the alias tuple $[\langle *\mathtt{q},\mathtt{a}\rangle,\langle *\mathtt{q},\mathtt{a}\rangle]$ reach the exit of statement 5. Finally, the alias tuples $([\phi,\langle *\mathtt{q},\mathtt{a}\rangle], [\phi,\langle *\mathtt{r},\mathtt{a}\rangle])$ reach the exit of the call statement 10.

Statement 11 then kills alias tuple $[\phi,\langle *\mathtt{q},\mathtt{a}\rangle]$, and creates the alias tuple $[\phi,\langle *\mathtt{q},\mathtt{b}\rangle]$ which reaches the call at statement 14. Alias tuple $[\phi,\langle *\mathtt{r},\mathtt{a}\rangle]$ is preserved through statements 11-13 and reaches the call at statement 14. Alias tuple $[\phi,\langle *\mathtt{r},\mathtt{a}\rangle]$ will result in $[\langle *\mathtt{r},\mathtt{a}\rangle,\langle *\mathtt{r},\mathtt{a}\rangle]$ being propagated to the exit of statement 3. Then this is killed by statement 4. Alias tuple $[\phi,\langle *\mathtt{q},\mathtt{b}\rangle]$ will result in $[\langle *\mathtt{q},\mathtt{b}\rangle,\langle *\mathtt{q},\mathtt{b}\rangle]$ being propagated to the exit of statement 3. This results in alias tuple $[\langle *\mathtt{q},\mathtt{b}\rangle,\langle *\mathtt{r},\mathtt{b}\rangle]$ at the exit of statement 4, which reaches the exit of statement 5 as does $[\langle *\mathtt{q},\mathtt{b}\rangle,\langle *\mathtt{q},\mathtt{b}\rangle]$. Finally, alias tuples $([\phi,\langle *\mathtt{q},\mathtt{b}\rangle], [\phi,\langle *\mathtt{r},\mathtt{b}\rangle])$ reach the exit of the call statement 14.

Although alias tuples $[\langle *\mathtt{q},\mathtt{a}\rangle,\langle *\mathtt{r},\mathtt{a}\rangle]$ and $[\langle *\mathtt{q},\mathtt{b}\rangle,\langle *\mathtt{r},\mathtt{b}\rangle]$ both reach the exit of statement 5 in proc1, the reaching alias abstraction ensures that these aliases are mapped back only to the corresponding calling contexts; therefore, alias tuple $[\phi,\langle *\mathtt{r},\mathtt{a}\rangle]$ will be mapped back to the exit of statement 10, and alias tuple $[\phi,\langle *\mathtt{r},\mathtt{b}\rangle]$ will be mapped back to the exit of statement 14. Thus, the only fixed location aliased to $*\mathtt{r}$ at statement 13 is a with *FSAlias*, but *FIAlias* will find both a

```
1.      int *p,*q,*r;
2.      int a, b;

3.      void proc1()
        {
4.          r = q;
5.          *r = 1;
        }

6.      void main()
        {
7.          q = &a;
8.          p = q;
9.          *p = 1;
10.         proc1();
11.         q = &b;
12.         *q = 1;
13.         *r = 2;
14.         proc1();
        }
```

Figure 7: Example of differences between *FIAlias* and *FSAlias*

and `b` aliased to $*$`r`, since it combines all calling contexts. Note that $*$`r` is reported aliased to both `a` and `b` at statement `5` using either alias algorithm.

3 MOD$_C$ Schema

The MOD$_C$ schema defines a family of algorithms which solve for modification side effects to fixed locations at program points, parameterized by the type of aliasing algorithm used. Side effects reported are differentiated by fixed-location type: *global, local, dynamically-created,* and *non-visible.* In solving for modification side effects, the MOD$_C$ problem is decomposed into subproblems that are individually easier to solve than the monolithic problem. The problem decomposition assumes that context-sensitive alias information is available; it preserves calling-context information with the side effects for as long as possible. The MOD$_C$ schema is described for *FSAlias*, but any approximation of calling context may be used instead. If the pointer alias algorithm used is context-insensitive, then conceptually all calling contexts are mapped to one context; that is, there is no differentiation in side effects returned from a procedure to any individual call site and the multiple subproblems distinguished by *RA* shown in Figure 8 become a single subproblem.

As mentioned previously, the first pass of the algorithm solves for aliasing information, *Alias*. Given the results of this analysis, two related problems are calculated: (i.) *PMOD*, a procedure-level summary of context-sensitive modification side effects which can occur to fixed locations, and (ii.) *CMOD*, a set of modified fixed locations at each program point corresponding to a specific context. *CMOD* solutions can then be used to derive MOD information for program points, while *PMOD* solutions can be used to derive a procedure-level summary of modification side effects.

The decomposition of the MOD problem is pictured in Figure 8, where P is a procedure, RA is a calling context (i.e., a reaching alias) and n is a program point. The following brief description of each subproblem will be augmented in the next section by the corresponding data-flow equations. $Alias(n, RA)$ is the pointer-alias solution at statement n under calling context RA. *DIRMOD*(n) captures all object names which occur on the left-hand-side of the assignment at program point n (e.g., `*p=, v=`). At an assignment n, *CondLMOD* widens *DIRMOD*(n) to include the effects of aliasing; *CondLMOD* contains only fixed locations. *CondIMOD*(P, RA) summarizes *CondLMOD* information for each calling context RA over all assignment statements in procedure P. *PMOD*(P,RA) is formed from local *CondIMOD* information for P and *PMOD* information propagated from procedures called by P all under context RA, thus calculating both direct and indirect side effects of P. *CMOD* at a call site is constructed from *PMOD* of the called procedure, and at an assignment, from *CondLMOD* of that statement. Finally, *MOD* at a statement is constructed from *CMOD* by summarizing over all contexts, as is *MOD* for a procedure.

A comparison of our MOD$_C$ decomposition to that for FORTRAN is given in Appendix A. Recall that although the MOD$_C$ schema is described with calling-context information available, an easy transformation (i.e., folding all contexts together) yields a MOD$_C$ algorithm for use with context-insensitive alias methods.

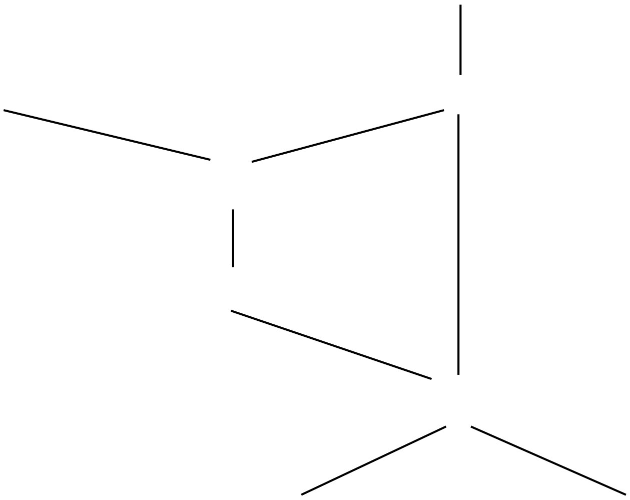


Figure 8: Decomposition of the MOD$_C$ problem

3.1 Data-Flow Equations

The following discussion makes these assumptions.

- *Assignment* is synonymous with *value-setting statement*; thus, **scanf** is considered an assignment.
- All object names are unique; thus the issue of *name hiding* is avoided. This can easily be met by appending object names with the function and file in which they are defined.
- *On bottom* data-flow information is computed (i.e., information at a statement incorporates the effects of that statement). Since a call statement is split into a call node and a return node in the internal representation, the information computed at the return node is "on bottom" information for the call statement, while the information computed at the call node is "on top" information for the call statement.
- The modification side-effects sets are associated with some representation of calling context to restrict attention to realizable paths; note that the MOD$_C$ schema is independent of the choice of calling-context abstraction.
- *Predecessors*(n) represents the set of predecessors of n in the ICFG.
- Trivial, reflexive aliases (e.g., $<*p,*p>$) are associated with the special reaching alias ϕ at all program points; this assumption simplifies the equation for *CondLMOD*. In the actual implementation these trivial aliases are not stored.

DIRMOD(n) is defined as the visible direct side effects at a statement; therefore, it requires no data-flow equation. *CondLMOD*(n,*RA*) is the set of fixed locations modified by the assignment at n because of aliases that occur on bottom of any of the predecessors of n under calling context *RA* for the procedure containing n:

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

If *DIRMOD*(n) is a fixed location, it is included in *CondLMOD*(n,ϕ) because reflexive aliases are associated with the special reaching alias ϕ.

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

$$CondIMOD(P, RA) = \bigcup_{n \text{ an assignment in } P} CondLMOD(n, RA) \tag{2}$$

PMOD(P, *RA*) is the set of fixed locations modified by procedure P, including the effects of calls from within P, considering only aliases corresponding to calling context *RA*. The *PMOD* sets for a procedure summarize its modification side effects for a given reaching alias. They are specified by the following, possibly recursive, system of data-flow equations which can be solved iteratively. The fixed point iteration used in the implementation is an optimistic lattice framework, which has been highly optimized with respect to the interprocedural transfer functions.

$$PMOD(P, RA) = \quad CondIMOD(P, RA) \cup \bigcup_{\substack{call_Q\ in\ P \text{ and} \\ RA' \,\in\, contexts\_of(call_Q, RA)}} \Big(b_{call_Q}(PMOD(Q, RA'))\Big) \tag{3}$$

In equation (3), $call_Q$ is a call site in P at which P calls Q. RA' represents the calling context induced by $call_Q$ during data-flow propagation in P under calling context RA of P. The function b_{call_Q}, specific to $call_Q$, maps object names from the called procedure (Q) to the calling procedure (P) according to scoping rules [CK87] and only returns fixed locations. Specifically, b_{call_Q} factors out all local fixed locations of Q (including formal parameters of Q), maps global fixed locations (including heap storage creation sites) to themselves, and maps $non_visible$s in Q to their corresponding fixed locations in P, which are either locals of P or $non_visible$s in P [LR92].[11]

It is possible with the MOD$_C$ schema to derive side effects at specific interesting statements, namely calls and assignments.

$$CMOD(n, RA) = \begin{cases} CondLMOD(n, RA) & \text{if } n \text{ is an assignment} \\ \displaystyle\bigcup_{RA' \in contexts\_of(n,RA)} b_n(PMOD(Q, RA')) & \text{if } n \text{ is a call of Q} \\ \emptyset & \text{otherwise} \end{cases} \qquad (4)$$

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

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

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

3.2 Example

The example in Figures 9, 10 and 11 shows a small C program analyzed by both the MOD$_C$($FSAlias$) and MOD$_C$($FIAlias$) algorithms.

For MOD$_C$($FSAlias$), both `main` and `R` are analyzed with reaching alias ϕ at their entries. The first call to `R` (shown by the solid line) creates the alias $\langle$`*b,x`$\rangle$ at the entry of `R`. The second call to `R` (shown by the dashed line) creates the alias $\langle$`*b,y`$\rangle$ at the entry of `R`. Procedure `R` is analyzed for each of these calling contexts. Note there are no aliases in `main`. The $FSAlias$ solution for `R` is shown in Figure 10; the $PMOD$ and $CMOD$ solutions computed are shown in the same figure. Empty entries in these tables mean either no alias or no side effect. Note that the entries in the tables indicate additions to the solution at a program point under a calling context. The whole solution at a point under a given context is the union of the entry in the table and the solution under calling context ϕ. Fixed location `b` is not in the solution for `main` because it's a local of `R`.

For MOD$_C$($FIAlias$), the $FIAlias$ solution is shown in Figure 11. The $PMOD$ solutions for `main` and `R` are shown as well. The $CMOD$ solution does differentiate side effects at program points within `main` and `R`, but notice that calling contexts are not differentiated.

[11] If context-insensitive aliases are used, there are no $non_visible$s.

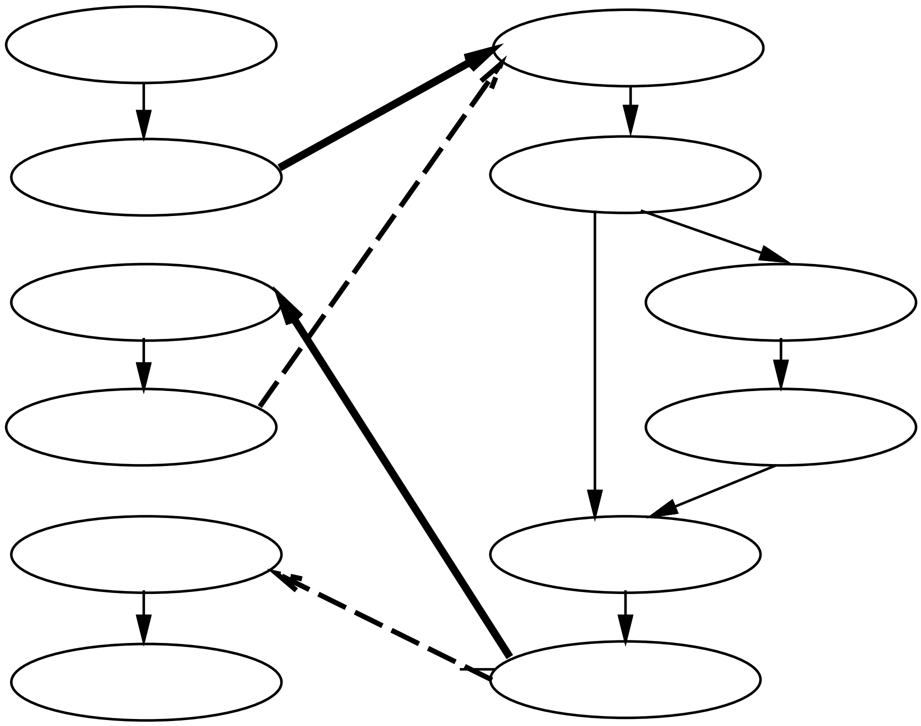


Figure 9: An example program and its ICFG

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

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

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

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

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

Figure 10: MOD$_C$($FSAlias$) solution for the example program of Figure 9

FIAlias solution for entire program
<*b,k>
<*b,x>
<*b,y>

PMOD Solution for main	*PMOD* Solution for R
{ x, y, k}	{ x, y, k, b}

CMOD Solutions for main						*CMOD* Solutions for R					
n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}	n_{11}	n_{12}
	{k, x, y }		{k, x, y }					{ b }	{ k, x, y }	{ k, x, y}	

Figure 11: MOD$_C$($FIAlias$) solution for the example program of Figure 9

3.3 Handling Call-by-reference Parameters

If a language uses call-by-reference parameter passing rather than the call-by-value parameter passing as in C, programs can be transformed into equivalent call-by-value programs by adding an additional level of indirection using the following transformation:

- For every call-by-reference formal `r` of type $\mathtt{t_r}$
  1. make `r` a call-by-value formal of type $\mathtt{t_r}$`*`
  2. replace `r` everywhere in the procedure with `(*r)`
- For every actual `a` corresponding to a call-by-reference formal
  3. replace `a` with `&(a)`; note that `a` is an arbitrary object name (e.g., `a[10]`, `p->next`)

When 2 and 3 above are both applicable, apply both. This means that a reference formal `r'` passed as an actual to another reference formal is transformed into `&(*r')`. Since semantically, dereference (`*`) and address (`&`) are effectively inverse operations[12] (`&*r'` $\equiv$ `*&r'` $\equiv$ `r'`), in the above case `r'` will be unchanged. This transformation is essentially the inverse of *refizing* as developed in [Car88]; for programs with single-level pointers it is equivalent to the transformation in [Lan92a] (p.50).

3.4 Worst-case complexity

The following definitions are useful in arguing the worst-case complexity of the MOD$_C$ schema.

- N_{alias} is the total number of aliases in the program.
- N_{assign} is the number of assignments in the program.
- N_{fixed} is the number of fixed locations.
- N_{ICFG} is number of nodes in the ICFG. This is roughly equivalent to number of program points.
- N_{proc} is the number of procedures in the program.
- C_{copy} is the cost of copying a set of fixed locations $[\Omega(N_{fixed})]$.
- C_{union} is the cost of the union operation over sets of fixed locations $[\Omega(N_{fixed})]$.
- E_{ICFG} is number of edges in the ICFG.
- M_{call} is the maximum number of calls for any one procedure.
- M_{pred} is the maximum number of predecessors of any assignment.
- M_{RA} is the maximum number of reaching aliases at the entry of any procedure.

The worst-case time complexity of a MOD$_C$ calculation is

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

Nevertheless, as for most static analyses, the worst-case time has little correlation with the observed behavior of the algorithm in practice.

To understand the sources of complexity in an algorithm, examine worst-case time complexity of each calculation in turn.

[12] In C, there is an exception: `&*(&x)` $\equiv$ `&x` but `*&(&x)` is illegal. The transformation will never cause this kind of translation.

- *DIRMOD*: $\mathcal{O}(N_{ICFG})$

- *Predecessors*: $\mathcal{O}(E_{ICFG})$

- *CondLMOD*: $\mathcal{O}(N_{alias} + N_{assign} * M_{RA} * M_{pred} * C_{union})$
  Access to the alias solution is necessary to compute *CondLMOD*, but each alias need only be considered once $[\mathcal{O}(N_{alias})]$. There are at most $(N_{assign} * M_{RA})$ *CondLMOD* sets and for each one at most $(M_{pred} + 1)$ unions are performed.

- *CondIMOD*: $\mathcal{O}(N_{assign} * M_{RA} * C_{union})$
  Each *CondLMOD*(n, RA), with n an assignment statement and RA a reaching alias, is conjoined into exactly one *CondIMOD*. Thus at most $(N_{assign} * M_{RA})$ unions are performed.

- *PMOD*: $\mathcal{O}(N_{proc} * M_{call} * M_{RA}^2 * N_{fixed}^2)$
  PMOD requires a fixed point calculation. There are at most $(N_{proc} * M_{RA})$ *PMOD* sets. *PMOD*(P, RA) is first initialized to *CondIMOD*(P, RA). For all procedures P and reaching aliases RA, this costs

  $$\mathcal{O}(N_{proc} * M_{RA} * C_{copy}) \tag{5}$$

  Secondly, *PMOD* is computed *once* for all procedures P and contexts RA using equation (3); this cost will be amortized over all calls in the program. There are $\mathcal{O}(N_{proc} * M_{call})$ calls. For a call to procedure Q, each set *PMOD*$(Q, RA^{'})$ will be considered at most M_{RA} times per union operation. There are at most M_{RA} such *PMOD* sets for Q. Therefore the cost of unions at a call is at most $(M_{RA} * M_{RA}) * (N_{fixed} + C_{union})$. The second term is the cost of the union as well as applying b_{call_Q} to each element. The cost of $contexts\_of$ and b_{call_Q} is negligible as these functions are already calculated by the alias calculation, and for the second pass are implemented as a simple (hash) table lookup. Including the cost of a union with *CondIMOD* in equation (3), the total cost of computing *PMOD* once is

  $$\mathcal{O}(\text{[number of calls] * [cost of unions per call] + [number of } PMOD\text{s] * [cost of one union]})$$

  Taking into account that $C_{union} = \Omega(N_{fixed})$ and the first term dominates, this is equal to

  $$\mathcal{O}([N_{proc} * M_{call}] * [M_{RA} * M_{RA} * N_{fixed}]) \tag{6}$$

  Finally the cost of the iteration must be counted. Each *PMOD* can change its value at most N_{fixed} times. Thus there are at most $N_{proc} * M_{RA} * N_{fixed}$ changes over all *PMOD*s. When *PMOD*$(Q, RA^{'})$ changes, *PMOD*(P, RA) of all the procedures P that contain a $call_Q$ such that $RA^{'} \in contexts\_of(call_Q, RA)$ must be recomputed. There are at most $M_{call} * M_{RA}$ such *PMOD*s. Thus, the cost of changing *PMOD*(P, RA) is $\mathcal{O}(C_{union} + N_{fixed})$: one union (we do not recompute equation (3) from scratch) plus the cost of applying b_{call_P} to each element of *PMOD*$(Q, RA^{'})$. The cost of the fixed-point iteration phase is $\mathcal{O}$([number of changes] * [recomputations per change] * [cost of one recomputation]) =

  $$\mathcal{O}([N_{proc} * M_{RA} * N_{fixed}] * [M_{call} * M_{RA}] * N_{fixed}) = \mathcal{O}(N_{proc} * M_{call} * M_{RA}^2 * N_{fixed}^2) \tag{7}$$

  Thus, the total cost for computing *PMOD* is the sum of equations (5), (6), and (7), which is $\mathcal{O}(N_{proc} * M_{call} * M_{RA}^2 * N_{fixed}^2)$.

- *CMOD*: $\mathcal{O}(N_{assign} * M_{RA} * C_{copy} + N_{proc} * M_{call} * M_{RA}^2 * N_{fixed})$
  For each assignment statement and each reaching alias, the cost is C_{copy}. For each call, $call_P$, (there are $\mathcal{O}(N_{proc} * M_{call})$ of them in the program) and each reaching alias, RA, b_{call_P} is applied to $PMOD(P, RA)$ and the results are conjoined. Thus, the cost for each $(call_P, RA)$ pair is $\mathcal{O}(M_{RA} * N_{fixed})$.

- *MOD*: $\mathcal{O}(N_{ICFG} * M_{RA} * C_{union})$

3.5 Counting Side Effects

An important issue in measuring the effectiveness of a data-flow analysis is the choice of an empirically observable metric by which to judge performance. The number of fixed locations reported experiencing side effects at an assignment, at a call, and for a procedure is the metric used in these experiments. This seems reasonable since it is of clear use in program understanding and compiling applications; if the number of measured side effects is too large at a program point, the analysis is not of practical use.

Some assignment statements in C involve *aggregate* types such as structs or unions. An aggregate is a fixed location whose fields can be simultaneously modified through *one* assignment. Arrays in C are not aggregates, because an array itself cannot be modified as one entity; all modifications occur through individual array elements.

Aggregates in MOD$_C$ solutions present problems in counting the numbers of fixed locations modified.

Suppose `s` is a struct type with fields `a, b` and `c`. It is possible that `s, s.a, s.b` and `s.c` all are modified by individual assignment statements and therefore all are found in a MOD$_C$ solution for a particular procedure. For example, in `main` in Figure 12 there are assignments to every field of struct `s3`. In procedure `p`, there is a struct assignment which simultaneously assigns to all three fields of struct `s1`. If struct fields are counted as fixed locations, then 3 side effects will be reported for `main` (one for each assignment); otherwise, 1 side effect to struct `s3` will be reported. Similar questions determine if 1 or 3 fixed locations will be reported as side effects for procedure `p` or if 1 or 3 fixed locations will be reported for procedure `r`. Note that the difference between `q` and `r` is that `r` has a struct assignment and a field assignment to the same struct, whereas `q` has a struct assignment and a field assignment to a different struct.

The MOD$_C$ implementation supports two counting schemes, called *Fields* and *NoFields*, respectively. To explain how aggregates are handled in the two counting schemes, refer to Figure 12. First an appropriate MOD$_C$ algorithm is applied to a program and sets of fixed locations collected at each assignment statement. For procedures like `r` both the entire struct and an individual field may occur in MOD(`r`) as a fixed-location side effect.

Counting fixed locations modified for an assignment statement is straightforward. For NoFields counting, any field name or struct name counts as 1 fixed location. For Fields counting, a struct assignment statement affects m fixed locations for a struct with m fields; the effect would be the same for an indirect access to the entire struct through a pointer to it. Each assignment to one field of a struct counts as 1 fixed location as well. These counts are illustrated in the Fields column in the code in Figure 12 for procedure `q`.

Counting becomes more complicated at a call when both a struct name and one of its field names are reported; this corresponds to the call of `r` in `main` in Figure 12. Intuitively, if Fields

```
                                            Number of Fixed Locations Reported Modified

                                            With Fields          With NoFields


struct s {int a, b, c;} s1, s2, s3;

/* s1:  struct-assignments only */
/* s2:  field-assignments only */
/* s3:  mixed assignments */

void p(){                                       3                    1
  s1 = s2;                                      3                    1
}

void q(){                                       4                    2
  s2.a = 4;                                     1                    1
  s3 = s1;                                      3                    1
}

void r(){                                       3                    1
  s3 = s2;                                      3                    1
  s3.a = 4;                                     1                    1
}

main(){                                         7                    3
  s3.a = 2;                                     1                    1
  s3.b = 3;                                     1                    1
  s3.c = 4;                                     1                    1
  p();                                          3                    1
  q();                                          4                    2
  r();                                          3                    1
}
```

Figure 12: Side effects counting example

counting is used, each distinct field is counted separately; thus the struct assignment statement in `r` finds 3 fixed locations experiencing side effects whereas the assignment to `s3.a` finds only 1. If NoFields counting is used, when a field name experiences a side effect it is as if the entire struct experienced the effect, so that the call to `r` reports 1 side effect (to struct `s3`), and the three assignments in `main` all are counted as causing a side effect to struct `s3`, rather than to its fields. Notice for the call of `q` in `main` only 2 fixed locations are reported as experiencing a side effect in NoFields counting, structs `s3` and `s2`. The same counting is used for $MOD(P)$ sets.

Unions are handled in much the same manner as structs. If any member of a union appears in a MOD_C set, under Fields counting it will be expanded to all of its members. The actual internal representation of fields/members for structs and unions in $FSAlias$ and $FIAlias$ uses a start position and length; this sometimes allows recognition of exact overlap of two struct fields (or members) and results in better precision in counting.

There is also a problem with Fields counting of dynamically created structs if a user creates their own *malloc*. Recall that a unique heap creation site name is created for all cells corresponding to a single allocation statement in the code. In Figure 13, procedure `a` only stores into struct `*s` and procedure `b` only stores into struct `*t`, yet 5 fixed locations are reported in $MOD($`a`$)$ and $MOD($`b`$)$, because the naming mechanism does not remember that the 5 fields come from two different structs. The problem is that the same heap creation site name for the *malloc* site in procedure `my-malloc` is used for all calling contexts. That name becomes associated with the sum of all the fields of all structs allocated using this procedure; effectively, encapsulation of the heap allocation within a procedure hides what's really going on from the aliasing algorithm.

NoFields counting is preferred and the results in Section 4 are reported using this scheme, though Figure 22 offers results using Fields counting and shows how this choice can influence the results reported.

4 Empirical Results

This section describes and discusses execution results of the MOD_C analyses. The MOD_C, $FSAlias$, and $FIAlias$ analysis code is implemented in C and analyzes a reduced version of C that excludes pointers to functions, exception handling, *setjump* and *longjump*, but allows type casting and unions. These results were gathered on a 75Mz processor *Sun Sparcstation 20* with 348 Mb of RAM and 527Mb swap space.

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

The last column of Table 1 indicates whether or not $FSAlias$ succeeds in calculating an alias solution for the program. $FSAlias$ is unable to calculate a solution for 8 of the programs because it runs out of virtual memory; $FIAlias$ can calculate a solution for all the programs. This raises the question of what characteristics of a program affect the availability of a flow-/context-sensitive alias solution. Certainly program size is a factor because of the relationship between size of solution

Program	LOC	ICFG Nodes	# Procs	# Calls	# Assigns		Flow-sens.
					All	Thru-deref	Alias Soln
allroots	215	422	8	20	72	3	√
fixoutput	401	617	7	13	123	5	√
diffh	1708	646	15	50	80	4	√
travel	862	698	16	24	170	3	√
ul	548	1027	15	36	168	6	√
plot2fig	1495	1077	27	78	159	16	√
lex315	719	1297	18	103	137	6	√
compress	1490	1319	16	29	274	11	√
clinpack	1226	1429	14	80	267	30	√
loader	1219	1563	31	80	242	78	√
mway	705	1576	22	43	406	71	√
ansitape	1596	1747	36	110	274	21	√
stanford	887	1771	48	80	369	42	√
pokerd	1243	1895	27	86	296	59	√
zipship	1283	1955	14	53	332	59	√
dixie	2109	2341	36	83	394	73	√
zipnote	3155	2407	20	71	348	86	√
learn	1483	2626	36	80	432	59	√
xmodem	1712	2672	28	156	447	97	√
compiler	2232	3008	39	350	304	2	√
zipcloak	3644	3033	30	93	424	104	√
sim	1439	3034	17	29	818	130	√
cdecl	1015	3196	33	204	448	25	√
diff	1708	3300	43	129	569	100	√
unzip	4106	3416	40	99	731	52	√
assembler	2673	3601	53	248	533	233	√
gnugo	2901	3651	29	89	650	109	√
livc	1886	4101	87	204	885	243	√
lharc	3303	4250	87	198	791	123	√
patch	2672	4608	56	271	750	135	√
simulator	3733	5574	100	409	666	107	√
arc	7507	5856	96	237	1105	160	√
triangle	1930	6119	19	43	1072	241	√
tbl	2511	6162	85	316	907	279	√
football	2222	7313	59	258	847	225	√
flex	6970	7376	86	307	1505	248	√
zip	7427	9288	109	324	1554	331	
072.sc	8087	13690	154	698	1826	201	
spim	19032	16740	168	974	1566	374	
larn	9546	21184	264	2218	2536	158	
tsl	14646	27302	450	2109	2350	587	
008.espresso	13567	30510	308	1830	5054	1524	
moria	24596	38572	432	3708	5893	1493	√
TWMC	23833	51627	204	796	10669	3949	
nethack	28735	58317	474	2837	4268	900	

Table 1: Program data set

```
struct S { int a, b, c; } x, *s;
struct T { float f, g; } y, *t;
/* x and y are initialized elsewhere in the program */

void a()
{  s = (struct S * ) my-malloc(sizeof(struct S));
   *s = x;
}

void b()
{  t = (struct T * ) my-malloc(sizeof(struct T));
   *t = y;
}

char *my-malloc(int size)
{  char *p = malloc(size);
   return p;
}
```

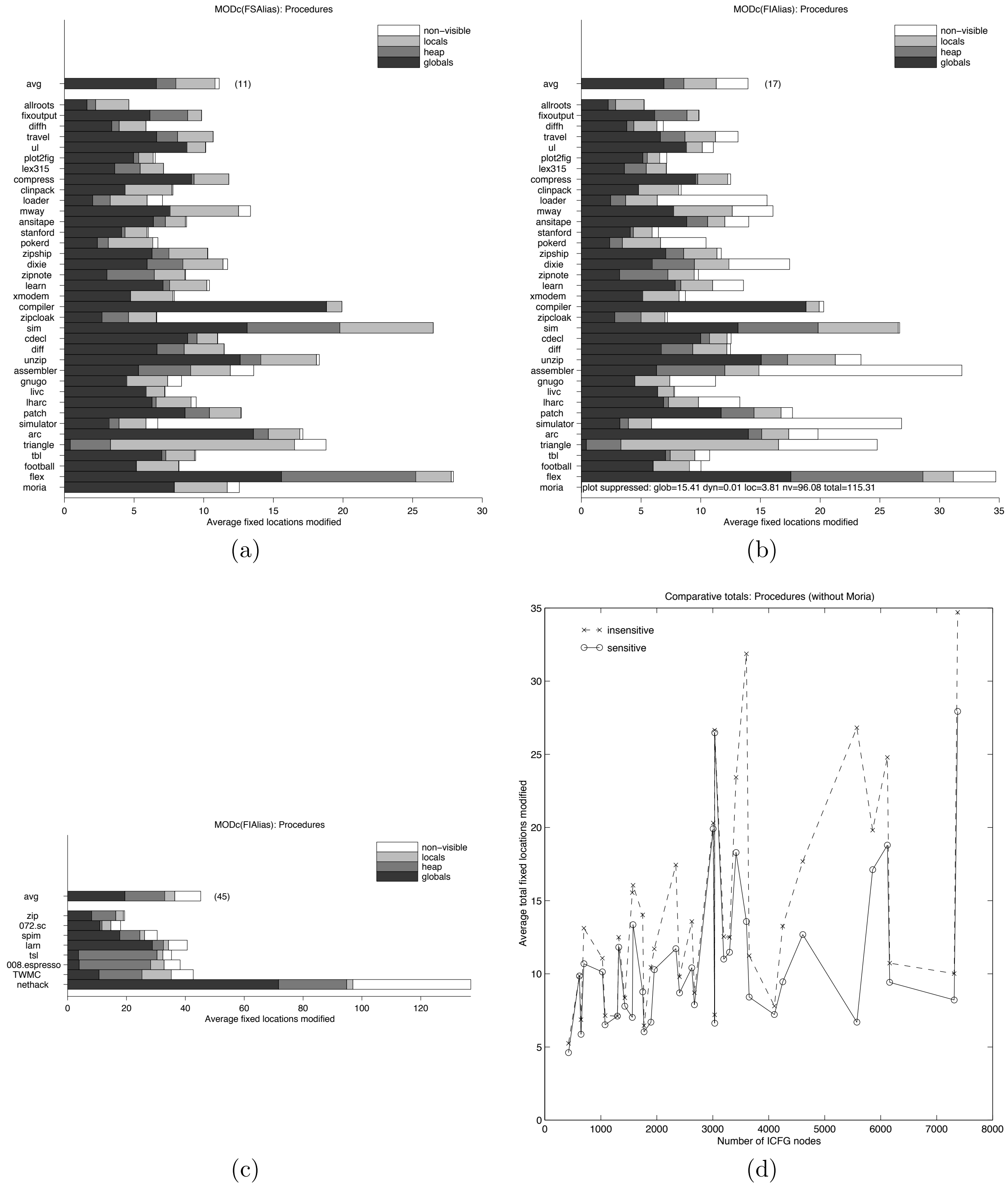
Figure 13: Problem with fields counting involving user-defined `malloc`.

and size of program, but it can not be the only factor as the contrasting results for *moria* and *zip* show. *Moria* is a "large" program with a *FSAlias* solution. Nor do the size differences between *zip* and *flex* seem vast enough to indicate a threshold on the power of *FSAlias*. The root of the problem is caused by recursive data structures. The *FSAlias* algorithm uses the somewhat naive *k-limiting* approximation to handle recursive data structures; however, in the cases of these larger programs that make excessive use of recursive data structures, the analysis gets bogged down in the generation and propagation of k-limited aliases. *Moria* has no recursive data structures. *Zip* uses recursive data structures much more heavily than *flex*.

MOD_C precision is reported in terms of the average number of fixed locations reported modified per kind of statement. The $\mathrm{MOD}_C(FSAlias)$ solution is always a subset of the $\mathrm{MOD}_C(FIAlias)$ solution, and both are safe, so that extra modifications reported by the $\mathrm{MOD}_C(FIAlias)$ solution are spurious. The raw data used to produce the figures in the following sections can be found in Appendix C.

4.1 Precision at Procedures and Call Statements

Figures 14(a) and 14(b) report the average numbers of fixed locations modified by procedures for both $\mathrm{MOD}_C(FSAlias)$ and $\mathrm{MOD}_C(FIAlias)$. The $\mathrm{MOD}_C(FIAlias)$ result for *moria* is elided because it skews the figure; the raw numbers are presented instead. The bars for each program are divided into the kind of location being modified. Each segment of the average bar is the average



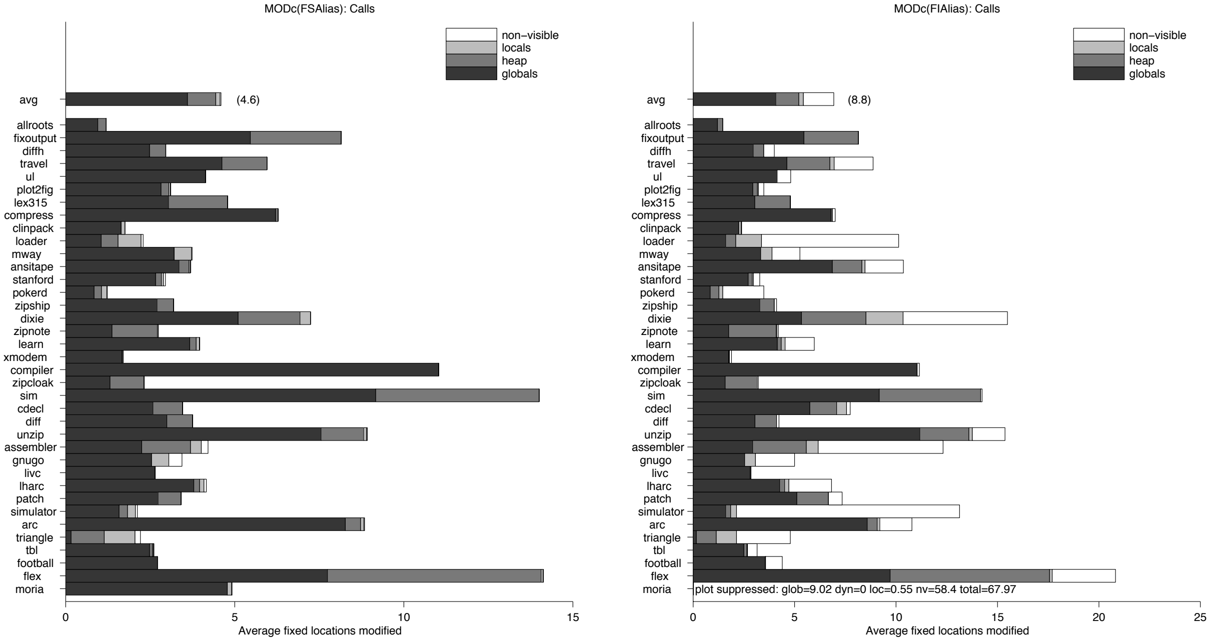
(a)

(b)

(c)

(d)

Figure 14: Average fixed locations modified by procedures.



(a) (b)

Figure 15: Average fixed locations modified by call statements.

over the numbers in Figure 14 for that kind of location. Formal parameters are reported as *locals*. Notice the enhanced precision of flow and context sensitivity (especially for the large program *moria*); MOD$_C$($FSAlias$) reports 11 fixed locations modified on average by procedures, against the 17 fixed locations reported by MOD$_C$($FIAlias$)[13]. Also notice that the average number of fixed locations modified is not closely correlated to program size.

Figure 14(c) shows the same results for the programs with only a flow-/context-insensitive MOD$_C$ solution. Figure 14(d) compares the average totals from Figures 14(a) and 14(b) in a more visually apparent manner. Programs are plotted by their sizes in ICFG nodes, along the x-axis. Again, *moria* is excluded from this comparison because it skews the figure.

Figures 15(a) and 15(b) present the same information as Figures 14(a) and 14(b) for fixed locations modified by call statements. The conclusions to be drawn are similar.

Figure 16 shows the standard deviations for the MOD$_C$($FSAlias$) results previously reported in Figure 14, (i.e., the number of fixed locations possibly modified).

Figure 17 addresses the comparative difference between the MOD$_C$($FSAlias$) and MOD$_C$($FIAlias$) solutions. If *sens* is the number of fixed locations reported modified at a program point by MOD$_C$($FSAlias$), and *insens* is the number reported by MOD$_C$($FIAlias$) at that same point, then the *relative mean* is the average of the calculation $(insens - sens)/insens$ over relevant program points in a program. Figure 17 shows the relative means for procedure side effects (Figure 17(a)) and call statement side effects (Figure 17(b)). These measurements indicate the proportion of the MOD$_C$($FIAlias$) solution that must be in error. Low numbers here mean that the

[13] Excluding *moria*'s result, MOD$_C$($FIAlias$) reports 14 fixed locations modified on average.

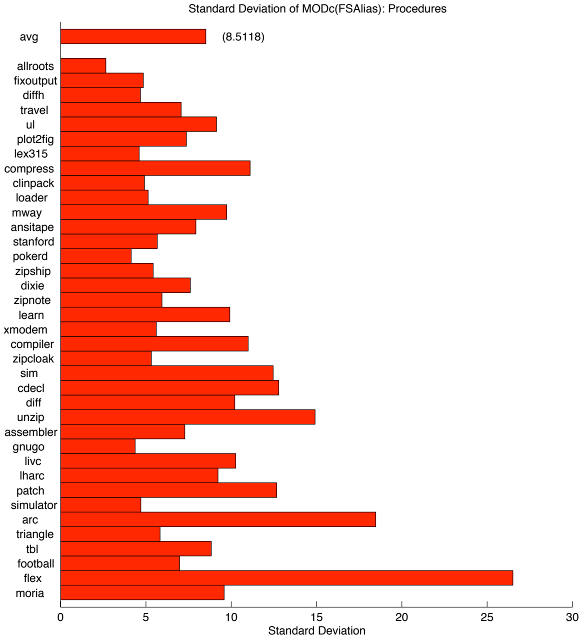
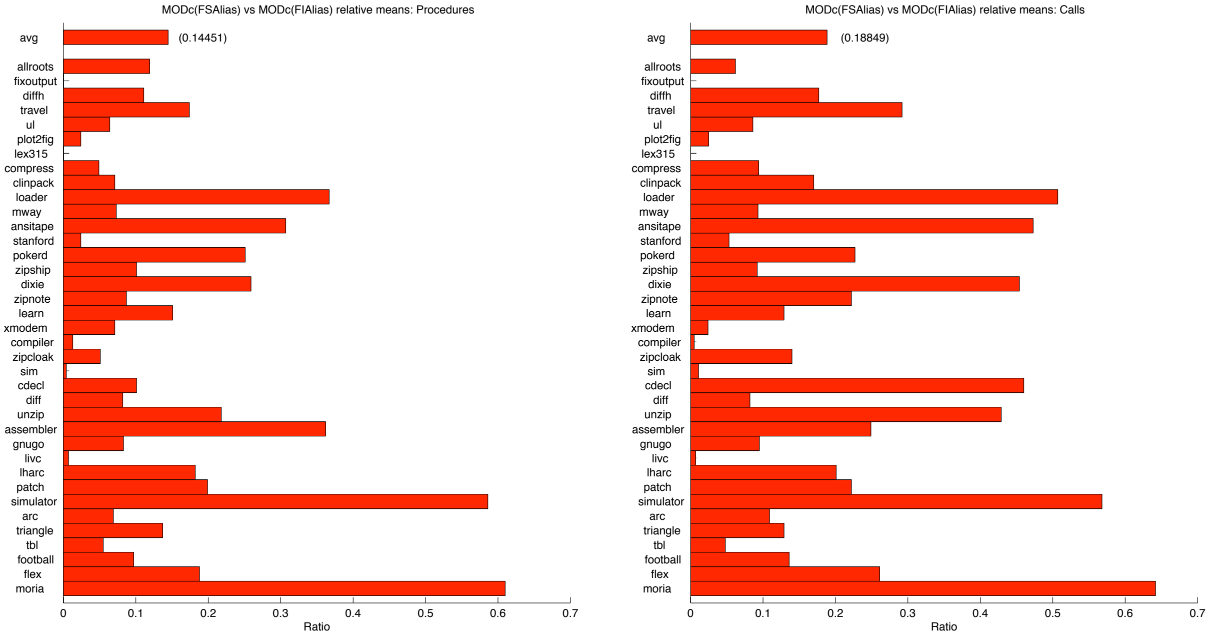
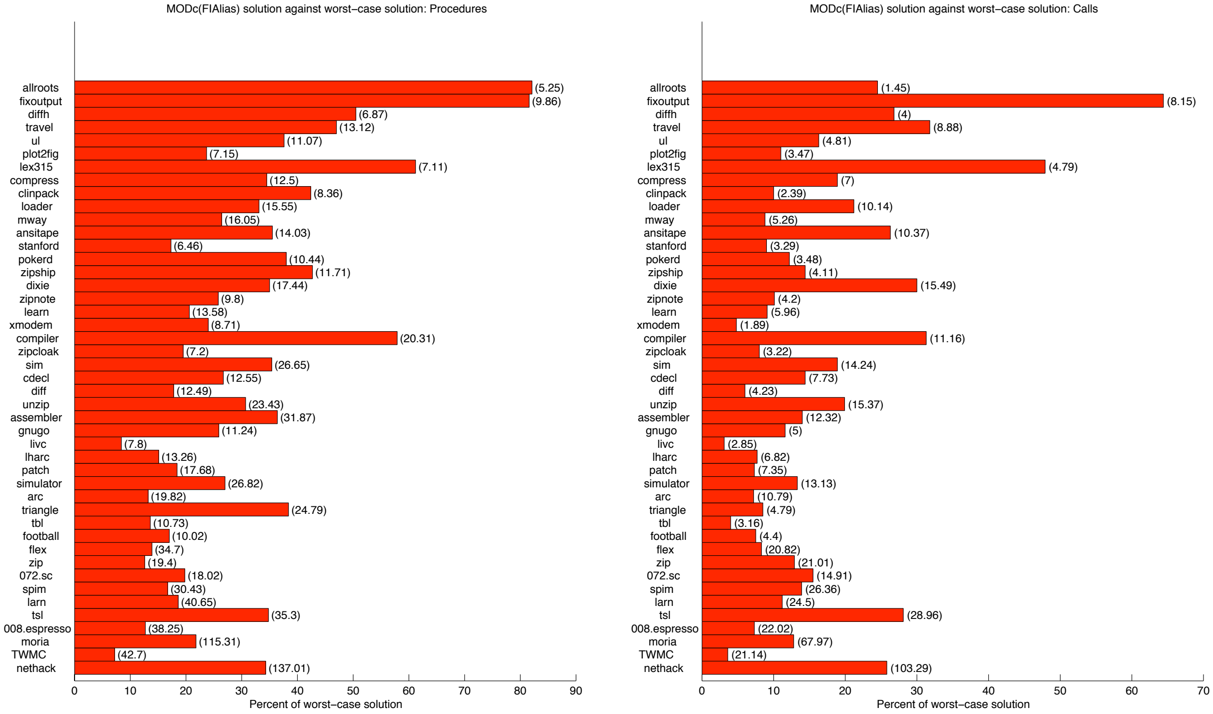


Figure 16: Procedure MOD$_C$($FSAlias$) standard deviations.



(a) (b)

Figure 17: Relative means for procedures and call statements.



(a) (b)

Figure 18: $\text{MOD}_C(FIAlias)$ Solution as percentage of worst-case solution.

$\text{MOD}_C(FIAlias)$ solution is nearly as precise as the $\text{MOD}_C(FSAlias)$ solution. Zero means that the solutions are the same. The average bars are calculated by treating all the program points in all programs as one set and averaging over all of them, rather than taking the average of the averages for each program. Also note that summing the relative means at each program point and averaging is substantially different from using the relative means calculation on the average total fixed locations modified results from Figures 14 and 15. For example, consider a procedure with two statements, where for the first statement $\text{MOD}_C(FSAlias)$ reports 1 location modified and $\text{MOD}_C(FIAlias)$ reports 2 locations modified, and for the second statement $\text{MOD}_C(FSAlias)$ reports 99 locations modified and $\text{MOD}_C(FIAlias)$ reports 100 locations modified. The average total locations modified by the procedure as reported by $\text{MOD}_C(FSAlias)$ is 50, and by $\text{MOD}_C(FIAlias)$ is 51. The relative mean for the first statement is .5, and the relative mean for the second is .01. The relative mean for the whole procedure is then .255, but the relative mean calculation using the average total numbers is 1/51 (a little less than .02). The numbers in Figure 17 are calculated using the former approach. These results give an indication of the trade-off in precision involved in using the flow-/context-insensitive analysis.

Even so, the precision of the $\text{MOD}_C(FIAlias)$ analysis is not to be understated. Figure 18 shows the average proportion of reported fixed locations modified by procedures and calls to the number of fixed locations potentially modified. The number of fixed locations potentially modified at a call or assignment statement is the sum of the number of globals in the program, the number of dynamic allocation sites in the program, the number of locals in the enclosing procedure, and the number of accessible non-visibles. The number of fixed locations potentially modified by a procedure is the sum of the numbers of potentially modified locations for each assignment and call statement in the procedure. Figure 18 shows what percentage the average totals reported by $\text{MOD}_C(\mathit{FIAlias})$ are of this worst-case. In parentheses after each bar is the average total number of fixed locations reported modified by $\text{MOD}_C(\mathit{FIAlias})$. Very low percentages indicate that the worst-case MOD_C solution is very much larger than what can be calculated using even flow-/context-insensitive data-flow analysis and indicate the significant advantages that inexpensive data-flow analysis can offer.

Table 2 presents another view comparing $\text{MOD}_C(\mathit{FSAlias})$ and $\text{MOD}_C(\mathit{FIAlias})$ that shows a frequency table of the numbers of procedures that modify a certain number of fixed locations. All the procedures in all the programs are considered together when constructing these frequency tables.[14] The frequency tables are broken down by fixed-location kind. For example, Table 2 (a) says that 367 of the procedures in all the programs modify 0 globals. The *Percent Below* column shows what percentage of the procedures have that many total side effects or more. Gaps in the sequence indicate that there are no procedures with that many side effects. The reports of procedures modifying 0 total fixed locations are not erroneous, but arise from simple procedures which have no local variables and no side effects (e.g., absolute value function).

The frequency tables for each kind of fixed location and for the totals seem to approximate a half normal distribution. For $\text{MOD}_C(\mathit{FSAlias})$, the frequency table for globals is the most spread out. This is probably an artifact of how globals are used in the data programs. For $\text{MOD}_C(\mathit{FIAlias})$, the frequency table for non-visibles is the most spread out. The $\text{MOD}_C(\mathit{FIAlias})$ frequency tables for all types of fixed locations apart from non-visibles appear to be close to the shape of the frequency tables for $\text{MOD}_C(\mathit{FSAlias})$, albeit flatter and longer, but the $\text{MOD}_C(\mathit{FIAlias})$ frequency table for non-visibles is much worse. Since $\text{MOD}_C(\mathit{FSAlias})$ is safe, this cannot be an artifact of the real solution. It appears that when the $\text{MOD}_C(\mathit{FIAlias})$ solution gets overly approximate, it is with regard to non-visibles. Why this is so, and what methods should be adopted to remedy the situation, are open questions.

4.2 Precision at Through-dereference Statements

Figures 19(a) and 19(b) show the average numbers of fixed locations modified by through-dereference assignment statements, similarly to Figure 14. These results are not discussed at length here because they are more related to the choice of aliasing algorithm than the MOD_C algorithm. Nevertheless, their precision is interesting.

Any executable assignment in a normally terminating program will modify at least one fixed location. Thus, 1 is a lower bound of total fixed locations modified per assignment statement (the dotted lines in Figures 19(a) and 19(b) show the line $x = 1$). The precision of these results for $\text{MOD}_C(\mathit{FSAlias})$ is very encouraging, and highlights the precision of the flow-/context-sensitive algorithm. The totals are all close to 1 with a maximum value of 2 and an average of 1.3. In contrast, $\text{MOD}_C(\mathit{FIAlias})$ is more imprecise, averaging 5.1, and sometimes being wildly inaccurate as in such cases as *moria*.[15] *Moria* is a large program with very many large (though non-recursive) data structures with several aliases. Perhaps the inability of the $\text{MOD}_C(\mathit{FIAlias})$ algorithm to

[14] The programs with no *FSAlias* solution are omitted when calculating the $\text{MOD}_C(\mathit{FIAlias})$ frequency table.

[15] Without *moria* the average total for $\text{MOD}_C(\mathit{FIAlias})$ is 2.6.

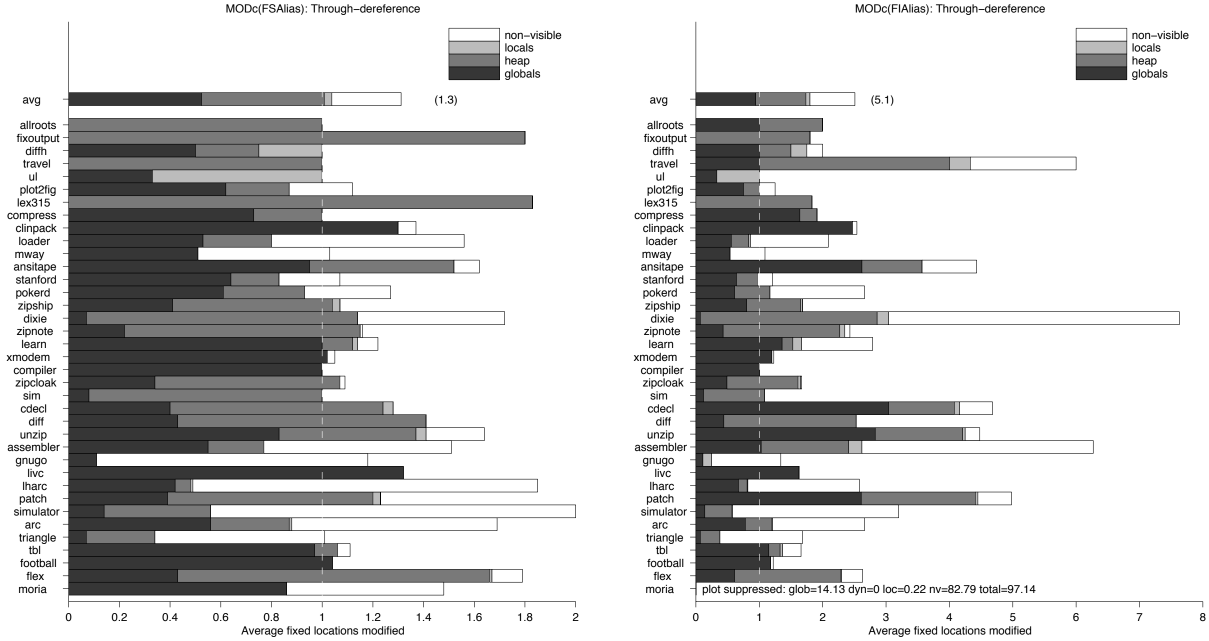
# Side Effects	Glob	Dyn	Loc	Nv	Total	Percent Below
0	367	1389	577	1624	132	100.00
1	422	117	348	87	224	92.77
2	175	119	239	32	181	80.49
3	181	66	160	25	133	70.58
4	133	31	132	19	139	63.29
5	51	9	79	4	104	55.67
6	64	5	60	9	120	49.97
7	34	12	48	1	65	43.40
8	50	9	37	3	58	39.84
9	21	2	29	3	69	36.66
10-12	53	28	57	6	123	32.88
13-15	28	7	27	2	98	26.14
16-18	38	18	14	2	54	20.77
19-21	35	3	8	1	43	17.81
22-24	22	2	5	2	35	15.45
25-27	14	1	0	3	27	13.53
28-30	14	1	1	0	34	12.05
31-33	10	0	0	0	27	10.19
34-36	12	0	2	0	12	8.71
37-39	4	2	0	1	18	8.05
40-42	3	0	2	0	12	7.07
43-45	12	0	0	0	12	6.41
46-48	38	0	0	0	17	5.75
49-51	1	0	0	0	11	4.82
52-54	6	0	0	0	10	4.22
55-57	1	1	0	0	8	3.67
58-60	5	0	0	0	7	3.23
61-63	2	0	0	0	5	2.85
64-66	5	0	0	1	9	2.58
67-69	3	0	0	0	9	2.08
70-72	0	0	0	0	3	1.59
73-75	2	3	0	0	1	1.42
76-78	6	0	0	0	3	1.37
79-81	2	0	0	0	3	1.21
82-84	2	0	0	0	0	1.04
85-87	3	0	0	0	6	1.04
88-90	0	0	0	0	2	0.71
94-96	0	0	0	0	1	0.60
106-108	2	0	0	0	0	0.55
109-111	1	0	0	0	0	0.55
115-117	0	0	0	0	1	0.55
118-120	0	0	0	0	1	0.49
121-123	0	0	0	0	1	0.44
124-126	0	0	0	0	2	0.38
127-129	0	0	0	0	1	0.27
145-147	0	0	0	0	1	0.22
151-153	3	0	0	0	0	0.16
226-228	0	0	0	0	3	0.16

MOD$_C(FSAlias)$
(a)

# Side Effects	Glob	Dyn	Loc	Nv	Total	Percent Below
0	351	1344	577	935	113	100.00
1	375	92	348	117	216	93.81
2	143	96	239	76	172	81.97
3	172	101	160	71	87	72.55
4	121	59	132	33	84	67.78
5	48	12	78	29	69	63.18
6	59	9	60	44	64	59.40
7	30	17	49	33	34	55.89
8	35	9	37	45	36	54.03
9	32	5	29	19	44	52.05
10-19	132	66	100	83	278	49.64
20-29	151	8	12	117	164	34.41
30-39	74	1	2	20	110	25.42
40-49	19	2	2	21	63	19.40
50-59	12	1	0	1	44	15.95
60-69	44	0	0	6	21	13.53
70-79	10	3	0	7	19	12.38
80-89	8	0	0	5	18	11.34
90-99	0	0	0	0	10	10.36
100-109	5	0	0	3	4	9.81
110-119	1	0	0	2	5	9.59
120-129	0	0	0	2	3	9.32
130-139	0	0	0	0	3	9.15
140-149	0	0	0	28	4	8.99
150-159	3	0	0	5	1	8.77
160-169	0	0	0	1	0	8.71
170-179	0	0	0	36	21	8.71
180-189	0	0	0	7	10	7.56
190-199	0	0	0	3	1	7.01
200-209	0	0	0	7	21	6.96
210-219	0	0	0	3	18	5.81
220-229	0	0	0	3	6	4.82
230-239	0	0	0	2	7	4.49
240-249	0	0	0	3	6	4.11
250-259	0	0	0	4	6	3.78
260-269	0	0	0	2	0	3.45
270-279	0	0	0	1	2	3.45
280-289	0	0	0	1	4	3.34
290-299	0	0	0	0	2	3.12
300-309	0	0	0	0	2	3.01
310-319	0	0	0	6	2	2.90
320-329	0	0	0	2	1	2.79
340-349	0	0	0	1	0	2.74
350-359	0	0	0	18	3	2.74
360-369	0	0	0	6	3	2.58
370-379	0	0	0	4	2	2.41
380-389	0	0	0	5	0	2.30
390-399	0	0	0	2	0	2.30
400-409	0	0	0	3	0	2.30
410-419	0	0	0	3	16	2.30
420-429	0	0	0	0	1	1.42
430-439	0	0	0	0	3	1.37
440-449	0	0	0	0	6	1.21
450-459	0	0	0	0	8	0.88
470-479	0	0	0	0	2	0.44
480-489	0	0	0	0	2	0.33
490-499	0	0	0	0	1	0.22
510-519	0	0	0	0	3	0.16

MOD$_C(FIAlias)$
(b)

Table 2: Procedure frequency tables.



(a) (b)

Figure 19: Average fixed locations modified by through-dereference assignments.

distinguish calling context and its inability to kill aliases explains the massive distortion between the $\mathrm{MOD}_C(FSAlias)$ and $\mathrm{MOD}_C(FIAlias)$ solutions for *moria*.

4.3 Discussion of Precision

A safe MOD_C solution is traditionally defined to be a point-wise superset of the precise solution. In other words, any fixed location that can be modified by some execution is guaranteed to be in the computed solution. The equations in Section 3.1 produce a safe solution. A *reverse-safe* solution is one that is point-wise a subset of the precise solution. In [Bar78], Barth defines the concept *precise up to symbolic execution* to mean precise assuming that all program branches are executable; effectively, this means that all intraprocedural paths are considered to be executable. It is possible to define *reverse-safe up to symbolic execution*. This is done by ensuring the approximate solution is a point-wise subset of the precise-up-to-symbolic-execution solution.

There are two major sources of imprecision. The first is due to intraprocedural paths that cannot be executed. The second is due to various interactions of aliases that do not occur on the same execution. We bound the latter type of imprecision but ignore the former, as it is very difficult to bound.

The equations in Section 3.1 can be extended to associate with each element of the $\mathrm{MOD}_C(FSAlias)$ either *yes* or *maybe* so that the part of the solution associated with *yes* is reverse-safe up to symbolic execution and the entire solution (i.e., the solution ignoring the *yes*/*maybe* information) is safe. Appendix B extends the equations in Section 3.1 in this manner. The safe solution implied by the modified equations is identical to the solution implied by the original equations.

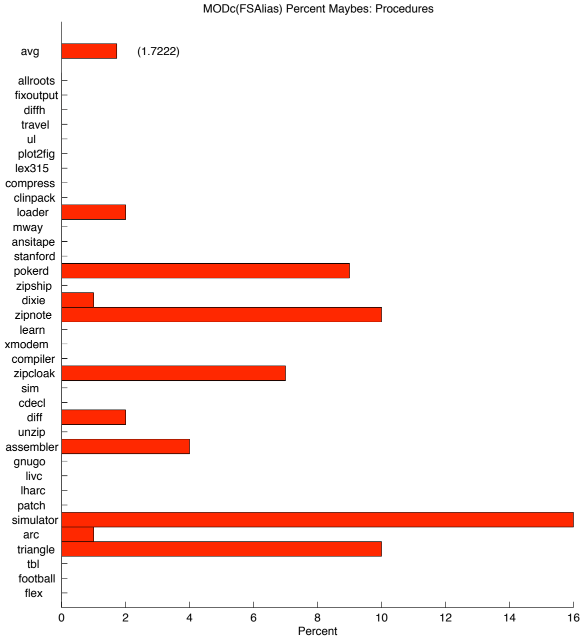


Figure 20: Percentage of average number of modifications that are *maybe*s

The equations in Appendix B allow a bound on the imprecisions to be computed. All $\text{MOD}_C(FSAlias)$ information not in the precise-up-to-symbolic-execution solution *must* be labeled with *maybe*. Thus, if P percent of the solution is labeled with *maybe* then at most P percent of the solution can be imprecise. Figure 20 is a bar chart of the percentage of $\text{MOD}_C(FSAlias)$ labeled with *maybe* for results for procedures, whose data imply that a majority of solutions on this data set are precise up to symbolic execution as none of the solution is associated with *maybe*. Even for the programs that have some of their solution associated with *maybe*, that part of the solution is small (16% maximum). The charts for through-dereference assignments and calls are similar in appearance to Figure 20 with averages of 4.1% and 2.4%, respectively. Computing $\text{MOD}_C(FSAlias)$ with *yes*/*maybe* information is more costly and could not be done for *moria*.

4.4 Cost of keeping calling contexts

The equations in Section 3.1 are context-sensitive in that they keep track of calling contexts in order to increase precision. This, of course, has a cost and that cost is maintaining the same MOD_C information for different contexts. The ratio of the size of *CMOD* plus the size of *PMOD* to the size of *MOD* indicates the percent of the *MOD* solution that is represented redundantly under various calling contexts. Figure 21 gives this ratio for the programs in the data set for which the $\text{MOD}_C(FSAlias)$ algorithm finds a solution. A ratio of 1.05 means that 5% of the solution is redundant. The average for these programs is around 1.01. Given the nature of Equations (3) and (4), this ratio should also be a good predictor of the extra time required to maintain the calling

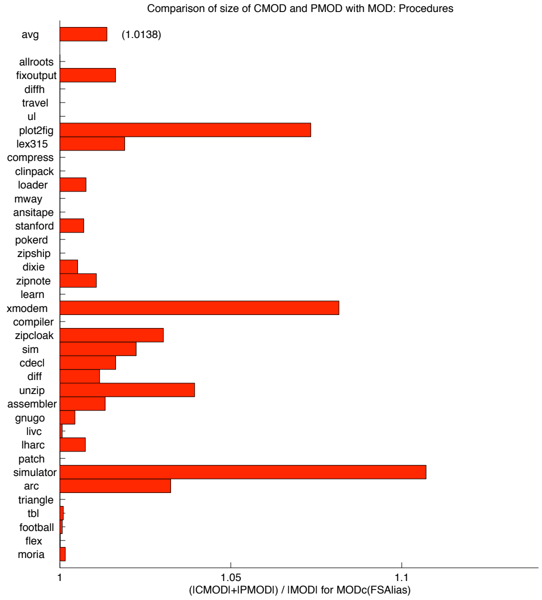


Figure 21: Cost of keeping calling contexts

contexts.

4.5 Effects of Counting

Section 3.5 explains the issues in determining the size of the *MOD* solution. Figure 22 compares the two methods of counting aggregates presented in that section for the $\text{MOD}_C(FSAlias)$ solution at procedures. Let *base* be the number of modifications determined in the manner used throughout this paper (i.e., NoFields counting). The sum of *base* and *extra* is the number of modifications that would be determined by counting each part of an aggregate separately. What is important about this figure is that the numbers are very different depending on how modifications are counted. In order to compare work by different research groups, minimally the assumptions used while counting must be documented. It would be even more useful if the same method of counting was used by all research groups.

4.6 Lines of best fit (Regressions)

Regressions on the data in this paper are potentially interesting as they can indicate how various factors are related (e.g., how the size of the program correlates to the number of modifications). However, there are two major potential problems:

1. the data set in this paper is too small to make strong conclusions, and

2. the regressions might not be good.

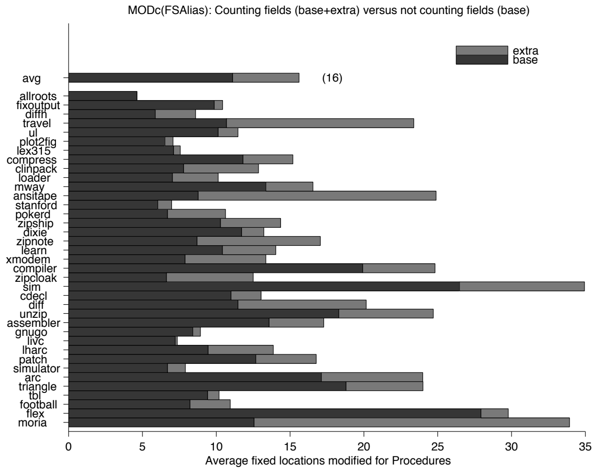


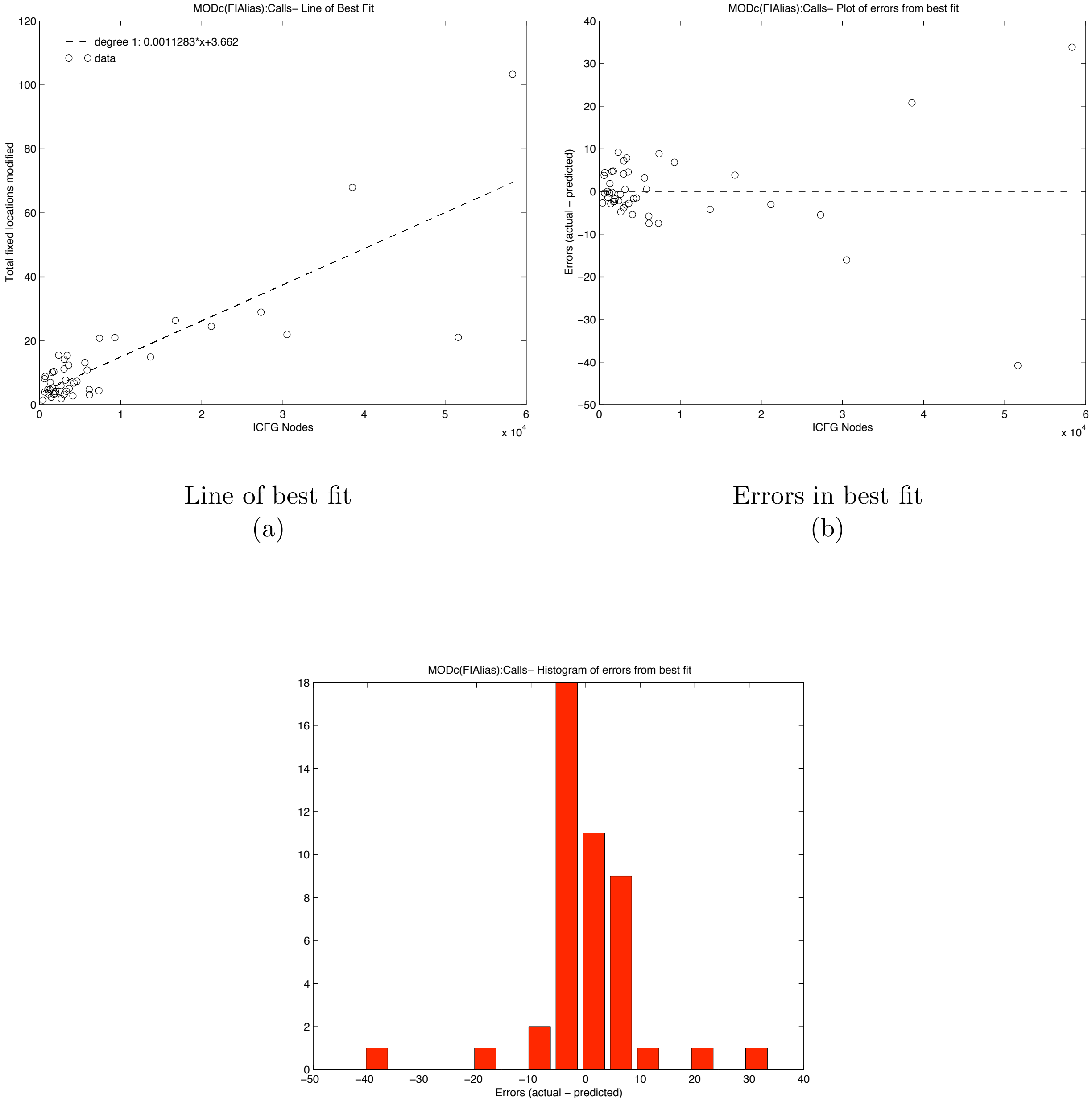
Figure 22: Different methods of counting

The former point can only be dealt with by expanding the data set vastly (at least an order of magnitude). Regressions of many of the figures in this paper look reasonable. The regression in Figure 23 (a) is one that appears good and is fairly typical. Given a line of best fit $y = mx + b$ for the data $\{(x_i, y_i)\}$ where x_i is program size in ICFG nodes and y_i is the program property being measured, an error factor e_i can be determined for each i such that $e_i = y_i - mx_i + b$. For example in Figure 14, y_i is the number of fixed locations reported on average per procedure.

The regression line indicates that there is a slight correlation between program size and the number of modifications. An additional 886 ICFG nodes (i.e., program statements) increases the expected number of modifications by 1 at each call site. However, the error factor seems to be a more important factor than the program size. This indicates that while size plays a role there are other, probably more significant, factors involved. For the data in this paper, this is what the regressions typically indicate. The regression in Figure 23 (a) is fairly good in that the plot of the errors (residuals) in Figure 23 (b) yields a fairly random pattern (although the magnitude of the error might get larger as program size gets larger) and the histogram of the errors in Figure 23 (c) is roughly normal.

4.7 Timing Results

Timing results are reported for the analysis times of the MOD$_C$ calculation, broken into its two passes, and for simple compilations of the programs in the data set using GNU's gcc compiler version 2.7.2 with no optimizations enabled and no linking. (All compile times reported in this section used this GNU compiler.) These numbers are as reported by the UNIX `time` utility, averaged over 5 executions. The notation MOD$_C$(FS) refers to the phase of the MOD$_C$($FSAlias$) algorithm *after* the alias solution has been computed; similarly for MOD$_C$(FI). Thus, these times *do not* include the alias analysis times, but are simply the time taken to calculate the MOD$_C$ solution given the alias solution. The total analysis time for, say, the flow-/context-sensitive analysis is the sum of

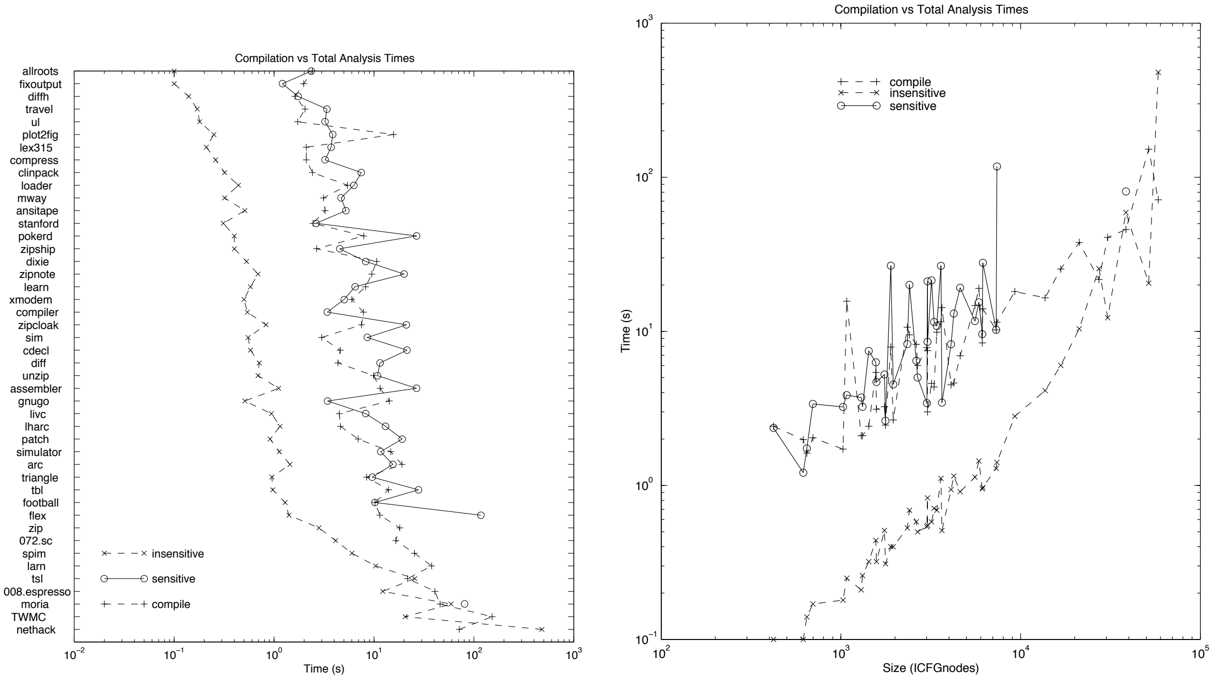


Line of best fit
(a)

Errors in best fit
(b)

Histogram of errors in best fit
(c)

Figure 23: A typical regression



(a) (b)

Figure 24: MOD$_C$($FSAlias$)time, MOD$_C$($FIAlias$)time, and compile time.

the $FSAlias$ time and the MOD$_C$(FS) time. The MOD$_C$ analysis time is dominated by the alias calculation.

Figure 24 (a) contrasts the MOD$_C$($FSAlias$) and MOD$_C$($FIAlias$) analysis times versus the compile times for the data programs, using a logarithmic scale for the time axis (x-axis). Figure 24 (b) plots the same data, but uses the program sizes (in ICFG nodes) as the x-axis showing the times on log scale on the y-axis. These figures demonstrate the dramatic difference between the sensitive and insensitive analyses, showing at least an order of magnitude difference between the two. The good news is that in most cases, the $FSAlias$ analysis time is comparable to the compile time. This is an important feature for any analysis destined for compiler optimization.

Figure 25 shows the times for the MOD$_C$(FS) and MOD$_C$(FI) phases of the MOD$_C$ calculation. MOD$_C$(FS) and MOD$_C$(FI) are both very fast, and are comparably fast, except in the case of *moria*. Notice that seemingly large differences are not large because the scale of the time axis is so small. The only explanation so far for *moria* is that the $FIAlias$ solution is sufficiently imprecise to cause the second pass of the MOD$_C$ algorithm to do significantly more work. Table 3 in Appendix C shows the raw data from which these figures are constructed.

4.8 Measurement of Memory Needed

The maximum memory needed by the two MOD$_C$ analyses was measured empirically by an external process while each analysis was running on each program in the data set. The memory needs of each algorithm include (i) a fixed-size of overhead space for algorithm data structures, (ii) the program

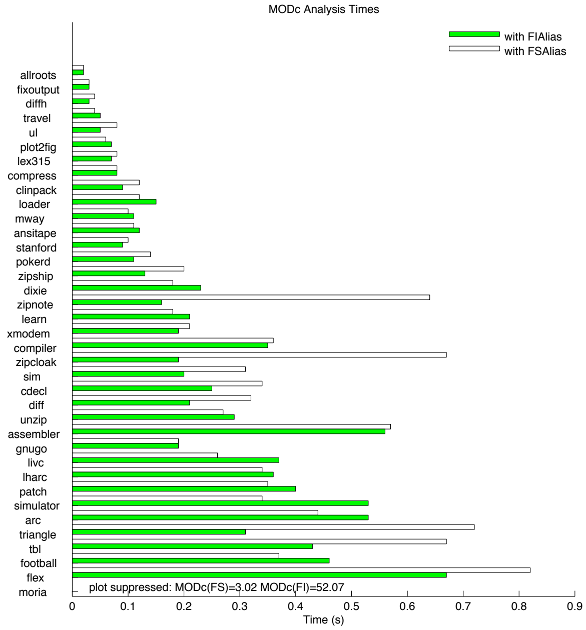


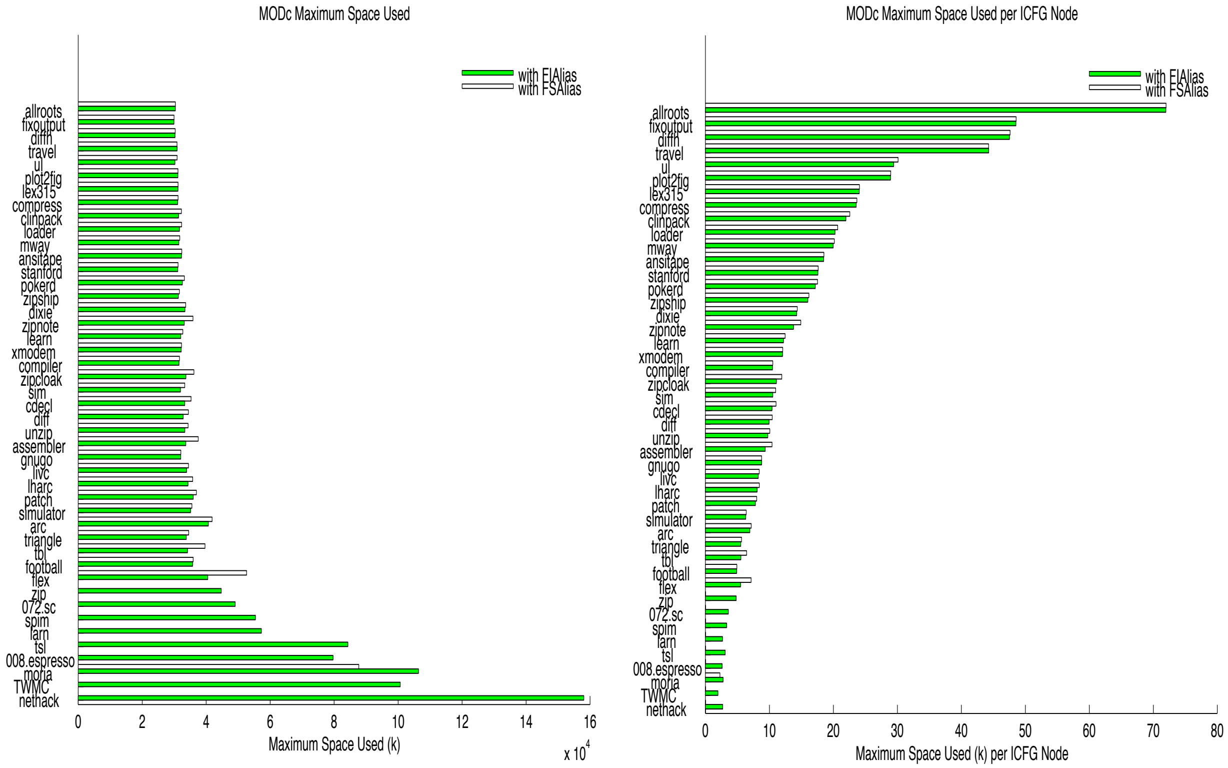
Figure 25: MOD$_C$(FS) and MOD$_C$(FI) analysis times.

representation, whose size varies linearly with program size and was the same for both analyses, and (iii) the analysis results. The maximum memory reported as used for each program in Figure 26 includes all three of these factors.

Figure 26(a) shows the absolute space needed, which varies for MOD$_C$($FIAlias$) from about 40Mb to almost 160Mb. The maximum space needed for MOD$_C$($FSAlias$), which runs out of virtual memory on the large programs, was about 90 Mb on *moria*. Note that the ratio of program size between *flex* and *moria* (as measured by number of ICFG nodes - see Table 1) is about 1:5.5 whereas the difference in maximum space needed for MOD$_C$($FSAlias$) is only about 50%. The maximum space needs of MOD$_C$($FIAlias$) seem to be growing precipitously; however, another way to view these results is shown in Figure 26(b), where the maximum space needed by each analysis is normalized by the number of ICFG nodes (i.e., approximately the number of C statements in the program). Here it is clear that the memory needs of MOD$_C$($FIAlias$) per statement of code are fairly stable for the larger programs (i.e., more than 20K LOC). Since no attempt has been made to optimize for memory usage in either analysis, these measurements are a “first cut” at what memory needs limit these analyses.

5 Related Work

Interprocedural Side-Effects Analysis. Interprocedural modification side effects were first handled by Allen for acyclic call multigraphs in FORTRAN programs [All74, Spi71]. Later, Barth explored the use of relations to capture side effects in recursive programs [Bar78]. Banning [Ban79]



(a) (b)

Figure 26: Memory usage of $\mathrm{MOD}_C(FSAlias)$ and $\mathrm{MOD}_C(FIAlias)$

first accomplished the decomposition of the MOD problem for FORTRAN (and other languages where aliasing is imposed only by call-by-reference parameter passing); he separated out two flow-insensitive calculations on the call multigraph: one for side effects and a separate one for aliases. The interprocedural side-effects problem for FORTRAN is flow-insensitive, but context-sensitive. Cooper and Kennedy [Coo85, CK88, CK87] further decomposed the problem into side effects on global variables and side effects accomplished through parameter passing. Burke showed that these two subproblems on globals and formals can be solved by a similar problem decomposition [Bur90]. All of this work targeted the programming model of FORTRAN77, a language without pointers. Choi, Burke, and Carini mention an interprocedural modification side-effects algorithm for languages with pointers based on their flow-sensitive pointer-aliasing analysis technique [CBC93, MLR$^+$93]; it is difficult to compare their work to this work, because they give no description of their algorithm and offer no implementation results.

Another approach to side-effect analysis is to perform an interprocedural pointer aliasing algorithm and then identify all variables experiencing side effects at indirect stores through a pointer (i.e., at through-dereference statements) using the aliases found [EGH94, GH98, Ruf95, HP98, SH97a, ZRL98, ZRL96]. This is often used as an empirical test of the precision of the alias solution obtained.

Related Analyses Related interprocedural analyses include compile-time interprocedural program slicing [GL91, HRB90, OO84, RR95, GS96, LH96, HC98, SHR99, TCFR96, Tip96, Ven91, Wei84, TAFM97, AG96, AG98], interprocedural def-use associations [PLR94, HS94, GH98, CR99] and, demand analyses [HRS95, DGS95, DGS96]. Slicing determines the data and control dependent parts of a program which correspond to a particular computation. Def-use associations trace value flow on static paths in a program; they are useful for various machine-independent optimizations and data-flow testing methods. Demand data-flow analysis seeks to efficiently answer queries about individual data-flow facts at a program point; a partial calculation is performed to derive the data-flow information, rather than a whole program analysis.

Interprocedural distributive finite subset problems can be solved using a graph reachability technique on an "exploded" call graph of the program [RHS95]. Capture of calling context is not an issue here since the problems being solved are of a form such that reachability in each procedure can be analyzed once for each parameter, regardless of calling context. The solution at a call site is obtained by using the parameter binding functions to identify incoming information with outgoing information at the corresponding return site. These relations between incoming and outgoing information are then memoized to avoid re-analysis. The underlying ideas of this analysis are related to notions expressed in the *conditional analysis* for aliasing due to single-level pointers [LR91]. Several solutions to MOD_C with different flow-insensitive, context-insensitive points-to approximation algorithms have been obtained by this method, since using program-wide aliases yields an approximate problem for MOD_C that yields the same side effects regardless of calling context [SH97a]. This Horwitz and Shapiro study shares the philosophy of the empirical results presented here, in that the effects of pointer aliasing on applications are reported. However, there are no flow- and/or context-sensitive analyses performed and direct comparison with $\text{MOD}_C(FIAlias)$ is difficult, since only a flow-/context-insensitive $MOD(P)$ is defined with no per-statement side effects, and indirect side effects to structure fields and union members are not distinguished.

Ruf [Ruf95] compared the effect of context sensitivity (or its lack) on a flow-sensitive points-to algorithm. Most of his reported data is with respect to the difference in precision of the points-to solution, with and without context information. Although his results were based on a possibly nonrepresentative benchmark suite (a fact observed by him), they indicated that no precision improvement was observed for his flow-sensitive algorithm on these programs by adding context sensitivity. For many reasons this study is difficult to compare with the results presented here. First, his VDG program representation is incomparable to the ICFG, not being a statement-level representation. Secondly, several independent structure field references may be merged, so that data on "average" number of side effects at a write statement may not be comparable to data at through-dereference statements. Thirdly, there is no flow-insensitive method in his paper.

Newer studies of the affect of flow sensitivity on points-to analysis [HP98, HP99] carefully track the performance of several flow-insensitive algorithms versus that of a flow-sensitive algorithm [BCCH94, BCCH97, CBC93, MLR$^+$93]. Comprehensive measurements of comparative precision at through-dereference statements (as well as precision on dereferenced reads), algorithm timings and memory usage are reported. These experiments on this data set showed that the precision of their flow-insensitive analysis was identical to that of their flow-sensitive analysis on 12 of their 21 benchmark programs [HP98]. Further work [Pio99] studies the affect of analysis precision on constant propagation. It is difficult to compare Hind and Pioli's results with those reported here, since all of their algorithms are context-insensitive and a different alias representation is used, which may alter the fixed location counts.

A recent technique uses SSA-form to transform the program in a way that effectively adds "adjustable" flow sensitivity to a flow-insensitive pointer-alias analysis [HH98]. No empirical results are given and it is not clear if this technique is scalable.

Pointer May_Alias Algorithms. This paper discusses a schema for finding side effects in C codes that is parameterized by the type of pointer aliasing technique used. Since the focus of this paper is the MOD$_C$ schema, the recent work in pointer analysis is merely summerized here. Recently, there have been many investigations of pointer aliasing algorithms which vary in cost and precision. Several concentrate on aliases in heap storage [HPR89, CWZ90, Deu94, HN90, EGH94, GH96a, GH96b, JM82a, LH88, SRW98]. Others calculate program-wide (flow-insensitive) aliases [Cou86, Gua88, BCCH94, And94, SH97b, Ste96b, Wei80, ZRL96, HP98]. There are flow-sensitive techniques as well which calculate program-point-specific aliases [Coo89, LR92, CBC93, MLR$^+$93, EGH94, GH98, WL95, Ruf95, HA90, SFRW90, CRL99]. Other work concentrates on aliases in higher order functional languages [Deu90, NPD87]. Newer work calculates pointer aliasing in programs with explicit concurrency achieved with co-begin, co-end constructs [RR99]. Another new paper offers a framework for normalizing the representation of structure fields, for use in determining aliasing in the presence of unions and casting [YHR99].

6 OBSERVATIONS

Recall that the empirical experiments refer to two specific implemented MOD$_C$ algorithms, MOD$_C(FSAlias)$ and MOD$_C(FIAlias)$. These observations are obtained from examination of these results and are specific to them. The obvious conclusion of the empirical results is that MOD$_C(FSAlias)$ yields significantly more precise solutions at far greater computation cost. Nevertheless, this is a complex and interesting trade-off.

6.1 Flow-/context-sensitive analysis

Flow-/context-sensitive data-flow analysis is capable of providing very accurate results for small, real world applications (10K LOC). As expensive as it is, the cost of sensitive analysis is still not prohibitive for a large subset of the data programs, being on the order of the time to compile the program. Thus, the $\mathrm{MOD}_C(\mathit{FSAlias})$ algorithm achieves scalability up to a certain point. Recall that previously published results were only up to 4700 lines of code. In particular, substantially larger programs that don't use certain program constructs heavily can be analysed; *moria* and *zip* vividly show the effects on the MOD_C analyses presented here of heavy use of (large) recursive data structures. This level of scalability is rather surprising for a program-point-specific analysis. Further, note that users of software-engineering tools such as data-flow testers or off-line program understanding databases which gather def-use information about a large program in order to query it later, may be willing to accept analysis costs of several times that of the compilation time and/or large memory needs. Nevertheless, it seems apparent that flow-/context-sensitive analysis is not going to scale to the next order of magnitude without a major innovation; whole-program flow-/context-sensitive analysis of large systems seems unattainable.

6.2 Flow-/context-insensitive analysis

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

6.3 Comparison of sensitivity

One claim being disputed in the analysis community is that flow-/context-sensitive analysis will obtain much better precision than flow-/context-insensitive analysis on important problems, such as modification side effects. The empirical results confirm the belief that sensitivity provides discernably increased precision in the solution obtained; for program transformation or validation applications, this accuracy may be required.

6.4 Where to now?

This study raises three topics for further exploration. The first is how to incorporate flow sensitivity into analysis of very large programs. Zhang *et al.* [ZRL98, ZRL96] report on a program decomposition strategy where the alias relation induces a partitioning of the assignment statements involving pointer variables. This in turn can be used to decompose the program into sections for which analyses of differing precision and cost can be applied; this is especially beneficial if an expensive analysis can be avoided where much greater accuracy will not be achieved. Initial experiments targeted recursive data structures as subjects for a flow-/context-insensitive alias analysis with appealing results. More experimentation is needed to comprehend the possibilities in this approach, both in terms of choice of analyses to apply to groups of program sections and of varying the program decomposition itself. Recent work focuses on reorganization and redesign of flow-/context-sensitive analysis to reduce memory costs of points-to analysis in C^{++} and Java; preliminary empirical findings are encouraging [CRL99, Cha99].

The second topic is how to make flow-/context-insensitive analysis more effective without increasing the cost. An interesting idea stems from the observation that safe analyses produce supersets of the precise solution. The intersection of the solutions generated by different, safe analyses for the same problem must also be safe, and may be closer to the precise solution. Recently, Shapiro and Horwitz used this idea with several flow-/context-insensitive points-to analysis algorithms [SH97b]. This approach needs more exploratory experimentation.

The final topic is to discern more fully the kind of program construct and programming style that foils data-flow analysis. Perhaps the availability of precise, flow-/context-sensitive data-flow analysis would be sufficient motivation to change programming practice, language design and programmers' habits. For instance, references in Java are a restricted form of pointers, and might be substantially easier to deal with under static analysis than C's general-purpose pointers.

7 CONCLUSION

This is the first interprocedural modification side-effects analysis for C (MOD_C) that obtains reasonable precision on programs with general-purpose pointer usage. The algorithm schema is parameterized by choice of pointer-aliasing method used as the first pass. Two MOD_C algorithms at opposite ends of the spectrum in terms of flow and context sensitivity were empirically profiled, with data collected for key statements (i.e., through-dereference assignments and calls) as well as for procedures (i.e., MOD(P)). This is the first empirical comparative study of the effects of both flow and context sensitivity in the context of an important data-flow problem. It is especially significant that the utility of the data-flow solution obtained is studied in an application context, because the hypothesis is that different applications will select different trade-offs in cost versus precision. A significant precision advantage was established for the flow-/context-sensitive side-effects analysis over that of the flow-/context-insensitive analysis; but, this performance was at a severe cost in analysis execution time usually of at least an order of magnitude. Maximum memory usage was tracked for these types of analyses but no meaningful comparison can be made with regard to memory usage until analyses have been optimized to save space.

Acknowledgements The authors acknowledge the contributions of Tom Marlowe to the comparative examples in Section 2 and Jyh-shiarn Yur to the description of *FSAlias* in Section 2. The authors also thank Atanas Rountev and Ulrich Kremer for their helpful discussions.

References

[AG96] D. Atkinson and W. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, 1996.

[AG98] D. Atkinson and W. Griswold. Effective whole-program analysis in the presence of pointers. In *Proceedings of the ACM SIGSOFT '98 Symposium on the Foundations of Software Engineering*, pages 46–55, November 1998.

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

[And94] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, 1994. Also available as DIKU report 94/19.

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

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

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

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

[BCCH94] Michael Burke, Paul Carini, Jong-Doek Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 234–250. Springer-Verlag, August 1994.

[BCCH97] 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.

[BH93] S. Bates and S. Horwitz. Incremental program testing using dependence graphs. In *Conference Record of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 384–396, January 1993.

[BR90] M. Burke and B. G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Transactions on Software Engineering*, 16(7), July 1990.

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

[Car88] M. D. Carroll. A new pointer-removing program transformation. Unpublished manuscript, 1988.

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

[Cha99] R. Chatterjee. *Modular Data-flow Analysis of Statically Typed Object-oriented Programming Languages*. PhD thesis, Department of Computer Science, Rutgers University, October 1999. in preparation.

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

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

[CK88] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 57–66, June 1988.

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

[Coo89] B. G. Cooper. *Ambitious Data Flow Analysis of Procedural Programs.* Master's thesis, University of Minnesota, May 1989.

[Cou86] D. S. Coutant. Retargetable high-level alias analysis. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 110–118, January 1986.

[CR88] M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute updates. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 274–284, January 1988.

[CR99] Ramkrishna Chatterjee and Barbara G Ryder. Data-flow-based testing of object-oriented libraries. Department of Computer Science Technical Report DCS-TR-382, Rutgers University, March 1999.

[CRL99] Ramkrishna Chatterjee, Barbara G. Ryder, and William. A Landi. Relevant context inference. In *Conference Record of the Twenty-sixth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, January 1999.

[CWZ90] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990. SIGPLAN Notices, Vol 25, No 6.

[Deu90] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 157–168, January 1990.

[Deu92] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the IEEE 1992 Conference on Computer Languages*, pages 2–13, April 1992.

[Deu94] A. Deutsch. Interprocedural may alias for pointers: Beyond k-limiting. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[DGS95] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Conference Record of the Twenty-second Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, January 1995.

[DGS96] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A demand-driven analyzer for data flow testing at the integration level. In *Proceedings of the International Conference on Software Engineering (ICSE'96)*, March 1996.

[EGH94] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–257, June 1994. Published as SIGPLAN Notices, 29 (6).

[Ema93] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, McGill University, Montreal, Canada, July 1993.

[FI98] P. Frankel and O. Iakounenko. Further empirical studies of test effectiveness. In *ACM SIGSOFT '98 Sixth International Symposium on the Foundations of Software Engineering*, pages 153–162, November 1998.

[FW93] P. Frankel and S. Weiss. An experimental comparison of the effectiveness of branch testing with data-flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[GH96a] R. Ghiya and L. Hendren. Connection analysis: A practical interprocedural heap analysis for c. *International Journal of Parallel Programming*, 1996.

[GH96b] R. Ghiya and L. Hendren. Is it a tree, a dag or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of the Twenty-third Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 1–15, January 1996.

[GH98] R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Conference Record of the Twenty-fifth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 121–133, 1998.

[GL91] K. Gallagher and J Lyle. Using program slices in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[GS96] R. Gupta and M. L. Soffa. Hybrid slicing: An approach for refining static slices using dynamic information. In *Proceedings of Third ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 29–40, 1996.

[Gua88] C. A. Guarna. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International Conference on Parallel Processing*, pages 212–220, 1988.

[HA90] W. L. Harrison III and Z. Ammarguellat. Parcel and Miprac: parallelizers for symbolic and numeric programs. In *Proceedings of International Workshop on Compilers for Parallel Computers*, pages 329–346. Ecole des Mines de Paris - CAI, UPMC - Laboratoire MASI, December 1990. Paris, France.

[HC98] M. J. Harrold and N. Ci. Reuse-driven interprocedural slicing. In *Proceedings of the 20th International Conference on Software Engineering*, pages 74–83, April 1998.

[HFGO94] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 191–200, May 1994.

[HH98] R. Hasti and S. Horwitz. Using static single assigment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 97–105, June 1998.

[HHN92] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving analysis and transformations of imperative languages. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260, June 1992. SIGPLAN Notices, volume 27, number 6.

[HN90] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transaction on Parallel and Distributed Systems*, 1990.

[HP98] M. Hind and A. Pioli. Assessing the effects of flow sensitivity on pointer alias analysis. In *Proceedings of International Static Analysis Symposiusm (SAS'98)*, pages 57–81. Springer-Verlag, September 1998.

[HP99] Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer alias analyses. Technical Report RC 21510, IBM T. J. Watson Research Center, March 1999.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 28–40, June 1989.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), January 1990.

[HRS95] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, October 1995.

[HS91] M. J. Harrold and M. L. Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65, March 1991.

[HS94] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[JM82a] N. D. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice Hall, 1982.

[JM82b] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 66–74, January 1982.

[Kil73] G. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, January 1973.

[KU77] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[Lan92a] W. Landi. *Interprocedural Aliasing in the Presence of Pointers*. PhD thesis, Rutgers University, January 1992. LCSR-TR-174.

[Lan92b] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, July 1988. SIGPLAN NOTICES, Vol. 23, No. 7.

[LH96] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, pages 495–505, March 1996.

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

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

[LRZ93] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[MLR$^+$93] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. Choi, M. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *ACM SIGPLAN Notices*, 28(9):67–70, September 1993.

[MR90a] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–196, January 1990.

[MR90b] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.

[MR91] T. J. Marlowe and B. G. Ryder. Hybrid incremental alias algorithms. In *Proceedings of the Twentyfourth Hawaii International Conference on System Sciences, Volume II, Software*, January 1991.

[NPD87] A. Neirynck, P. Panangaden, and A. Demers. Computation of aliases and support sets. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 274–283, January 1987.

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

[Ost90] Thomas J. Ostrand. Data-flow testing with pointers and function calls. In *Proceedings of the Pacific Northwest Software Quality Conference*, October 1990.

[Pio99] Anthony Pioli. Conditional pointer aliasing and constant propagation. Master's thesis, SUNY at New Paltz, 1999. Available at http://www.mcs.newpaltz/tr as Technical Report 99-102.

[PLR94] H. D. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[Pol88] C. D. Polychronopolous. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

[PS89] L. Pollock and M. Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12), December 1989.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, September 1994.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-second Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 49–61, January 1995.

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

[RP88] B. G. Ryder and M. C. Paull. Incremental data flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

[RR95] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineeering*, pages 41–52, 1995.

[RR99] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 77–90, June 1999.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995.

[Ruf97] E. Ruf. Partitioning data flow analysis using types. In *Conference Record of the Twenty-fourth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 15–26, January 1997.

[Ryd83] B. G. Ryder. Incremental data flow analysis. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 167–176, January 1983.

[SFRW90] S. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. A logic-based approach to data flow analysis. In *Proceedings of the Second International Workshop in Programming Language Implementation and Logic Programming*, pages 277–292, August 1990. Volume 456 of Lecture Notes in Computer Science.

[SH97a] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the Fourth International Symposium on Static Analysis (SAS'97)*, pages 16–34, September 1997.

[SH97b] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the Twenty-fourth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[Shi88] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, Jun 1988.

[SHR99] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering*, pages 432–441, May 1999.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

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

[SRW98] 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.

[Ste96a] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Proceedings of the Sixth International Confernce on Compiler Construction*, pages 136–150, April 1996. Also available as LNCS 1060.

[Ste96b] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the Twenty-third Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[TAFM97] P. Tonella, G. Antoniol, R. Fiutern, and E. Merlo. Flow-insensitive c^{++} pointers and polymorphism analysis and its application to slicing. In *Proceedings of the 19th International Conference on Software Engineering (ICSE97)*, pages 433–443, 1997.

[TCFR96] F. Tip, J-D Choi, J. Field, and G. Ramalingam. Slicing class hierarchies in c^{++}. In *Proceedings of OOPSLA'96: Conference on Object-oriented Programming Systems, Languages and Applications*, pages 179–197, October 1996.

[Tip96] Frank Tip. A survey of program slicing technques. *Journal of Programming Languages*, 3(3):121–189, 1996.

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

[Wei80] W. E. Weihl. *Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables*. Master's thesis, M.I.T., June 1980.

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

[Wey94] E. Weyuker. More experience with data flow testing. *IEEE Transactions on Software Engineering*, 19(9):912–919, September 1994.

[WL95] Robert Wilson and Monica Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995. also available as SIGPLAN Notices, 30(6).

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

[YHR99] S.H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 91–103, May 1999.

[Zha98] Sean Zhang. *Practical Pointer Aliasing Analyses for C*. PhD thesis, Rutgers University, August 1998. Also available as Dept of Computer Science Technical Report DCS-TR-367.

[ZRL96] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Proceedings of the 4th Annual ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 81–92, October 1996.

[ZRL98] Sean Zhang, Barbara G. Ryder, and William A. Landi. Experiments with combined analysis for pointer aliasing. In *Proceedings of ACM SIGPLAN Workshop on Program Analysis and Software Tools for Engineering*, pages 11–18, June 1998.

A Comparison with the MOD Decomposition for FORTRAN

The decomposition of the MOD problem for C in Figure 8 is similar in structure to the original decomposition for FORTRAN by Banning [Ban79], in the sense that both calculate local side effects in each procedure first, and then set up data-flow equations on call graphs to compute procedure-level side effects (i.e., a flow-insensitive interprocedural calculation).

The two decompositions are also similar in what is included in the MOD sets. In FORTRAN programs, variables are the only fixed locations and therefore various MOD sets in the decomposition for FORTRAN include just variable names. In C, pointers and dynamic allocation are allowed.

The two decompositions differ in their treatment of aliases. In the FORTRAN decomposition, aliases are computed at procedure calls. This is possible because for FORTRAN programs, only procedure calls can create aliases and aliases created by a call hold throughout execution of the procedure being called. In the MOD decomposition in Figure 8, aliases are computed at pointer assignments and procedure calls (i.e., at program points), because aliases vary intraprocedurally. An alias at a program point is associated with a reaching alias for the procedure containing that program point. These reaching aliases differentiate side effects caused by different calls of the same procedure.

The MOD problem for FORTRAN was further decomposed by Cooper and Kennedy [Coo85, CK88, CK87] into two subproblems, one on global variables and another on reference formals. The first subproblem is provably *rapid* [MR90b] while the second one is not. Similarly, for the C language, side effects on global fixed locations (including heap storage creation site names) can be separated from side effects on locals and $non_visible$s.

This can be done by introducing a new set $CondIMOD^{+}(P, RA)$ to the decomposition of Figure 8. This new set is the set of fixed locations either modified directly in procedure P or modified as $non_visible$s in procedures called by P, considering only aliases in P that are conditioned on RA. The $non_visible$s modified by procedures called in P are local variables of either P or other procedures that have called P, directly or indirectly. The system of data-flow equations for $CondIMOD^{+}$ sets is as follows, where b'_{call_Q} is the function b_{call_Q} in Section 3.1 restricted to mapping of $non_visible$s in Q to either locals or $non_visible$s in P.

$$CondIMOD^{+}(P, RA) = \; CondIMOD(P, RA) \cup \bigcup_{\substack{call_Q \;\in\; P \text{ and} \\ RA' \in contexts\_of(call_Q, RA)}} \Big(b'_{call_Q}(CondIMOD^{+}(Q, RA')) \Big)$$

After obtaining $CondIMOD^{+}(P, RA)$ solutions, the $PMOD$ sets can be computed by solving the following data-flow equations.

$$\begin{aligned} PMOD(P, RA) = \; & CondIMOD^{+}(P, RA) \cup \\ & \bigcup_{\substack{call_Q \; in \; P \text{ and} \\ RA' \in \; contexts\_of(call_Q, RA)}} \Big\{ obj \,\Big|\, obj \in PMOD(Q, RA') \text{ and } obj \text{ is global} \Big\} \end{aligned}$$

Figure 27 compares various MOD sets defined in this MOD decomposition for C and those in the decomposition for FORTRAN as presented in [CK87].

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

Figure 27: Comparison of MOD decompositions for FORTRAN and C

B Calculation of MOD with *maybe*s

In order to incorporate *maybe* information from the alias solution into the MOD solution, the following definitions are needed:

- $\mathcal{L}_1$ is the lattice $(\{maybe, yes\}, \sqsubseteq_1, \sqcup_1, \sqcap_1, \top_1, \bot_1)$ implied by the relation $maybe \sqsubseteq_1 yes$.
- F is the set of fixed locations of the program being analyzed.
- $\mathcal{L}$ is $(S, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$
    - $S = powerset(F \times \{maybe, yes\})$
    - $a \sqsubseteq b$ iff $[(\forall s)[s = (f, d_1) \in a] \Rightarrow [(\exists d_2) \text{ such that } (f, d_2) \in b \text{ and } d_1 \sqsubseteq_1 d_2]]$
    - $$a \sqcup b = \bigcup_{f \in F} \left(\begin{array}{ll} \{(f,d)\} & \left\{ \begin{array}{l} [(\exists d_1) \text{ such that } (f, d_1) \in a] \wedge \\ [(\exists d_2) \text{ such that } (f, d_2) \in b] \wedge \\ d = d_1 \sqcup_1 d_2 \end{array} \right. \\ & \\ \{(f,d)\} & \left\{ \begin{array}{l} (f, d) \in a \wedge \\ [(\nexists d_1) \text{ such that } (f, d_1) \in b] \end{array} \right. \\ & \\ \{(f,d)\} & \left\{ \begin{array}{l} (f, d) \in b \wedge \\ [(\nexists d_1) \text{ such that } (f, d_1) \in a] \end{array} \right. \\ & \\ \emptyset & \text{otherwise} \end{array} \right)$$
    - $\sqcap$ is analogous to $\sqcup$ but is not needed
    - $\top = F \times \{yes\}$
    - $\bot = \emptyset$
- *DIRMOD*, *Predecessors*, and b_{call_Q} as in Section 3.
- For all program points n and all reaching aliases RA
    - $(\langle x, y \rangle, yes) \in Calias'(n, RA)$ if x and y are definitely aliases on some path to n given RA reaches the entry of the procedure containing n. Notice, this is not must alias but simply a guarantee that $\langle x, y \rangle$ is in the precise up to symbol execution alias solution.
    - $(\langle x, y \rangle, maybe) \in Calias'(n, RA)$ if x and y may be aliases on some path to n given RA reaches the entry of the procedure containing n and the alias algorithm assumes it is for safety.
- For all program points n, $(\langle x, x \rangle, yes) \in Calias'(n, \phi)$ where $\langle x, x \rangle$ is any trivial, reflexive alias.
- $contexts\_of'$ is identical to $contexts\_of$ in Section 3.1 with *yes* or *maybe* associated with each alias as above.

The MOD problem can be decomposed when *yes* and *maybe* are to be associated with the solution in a manner similar to that in Figure 8.

First, notice that $obj \in CondLMOD(n, RA)$ should be associated with *yes* (*maybe*) iff the alias responsible for it is associated with *yes* (*maybe*).

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

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

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

A fixed location in *PMOD*'(P, RA) is definitely modified if it is definitely modified (associated with *yes*) in a called procedure *and* the alias at the call site which triggers that modification is also associated with *yes*:

$$\begin{aligned} PMOD'(P, RA) = \quad & CondIMOD'(P, RA) \cup \\ & \bigcup_{\substack{call_Q\ in\ P \text{ and} \\ (RA^{'}, b') \in\ contexts\_of'(call_Q, RA)}} \left\{ (obj, b) \;\middle|\; \begin{array}{c} (obj', b'') \in PMOD'(Q, RA^{'}) \wedge \\ obj \in b_{call_Q}(\{obj'\}) \wedge b = b' \sqcap_1 b'' \end{array} \right\} \end{aligned}$$

$CMOD'$ is simple for assignments and for procedure calls is analogous to $PMOD'$.

$$CMOD'(n, RA) = \begin{cases} CondLMOD'(n, RA) & \text{if } n \text{ is an assignment} \\ \mathcal{S}_{(n,RA)} & \text{if } n \text{ is a call of Q} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\text{where} \quad \mathcal{S}_{(n,RA)} = \bigcup_{(RA^{'}, b') \in\ contexts\_of'(n,RA)} \left\{ (obj, b) \;\middle|\; \begin{array}{c} (obj', b'') \in PMOD'(Q, RA^{'}) \wedge \\ obj \in b_n(\{obj'\}) \wedge b = b' \sqcap_1 b'' \end{array} \right\}$$

Finally, MOD' is (n is in procedure P)

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

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

Program	ICFG Nodes	Compile time (s)	$FSAlias$ time (s)	$FIAlias$ time (s)	MOD$_C$(FS) time (s)	MOD$_C$(FI) time (s)
allroots	422	2.42	2.34	0.08	0.02	0.02
fixoutput	617	1.98	1.18	0.07	0.03	0.03
diffh	646	1.62	1.70	0.11	0.04	0.03
travel	698	2.04	3.34	0.12	0.04	0.05
ul	1027	1.72	3.15	0.13	0.08	0.05
plot2fig	1077	15.68	3.80	0.18	0.06	0.07
lex315	1297	2.10	3.64	0.14	0.08	0.07
compress	1319	2.10	3.16	0.18	0.08	0.08
clinpack	1429	2.42	7.34	0.23	0.12	0.09
loader	1563	5.42	6.18	0.29	0.12	0.15
mway	1576	3.12	4.59	0.21	0.10	0.11
ansitape	1747	3.24	5.13	0.39	0.11	0.12
stanford	1771	2.46	2.53	0.22	0.10	0.09
pokerd	1895	7.90	26.52	0.29	0.14	0.11
zipship	1955	2.66	4.33	0.27	0.20	0.13
dixie	2341	10.64	8.09	0.30	0.18	0.23
zipnote	2407	9.50	19.40	0.53	0.64	0.16
learn	2626	8.24	6.27	0.37	0.18	0.21
xmodem	2672	6.00	4.81	0.31	0.21	0.19
compiler	3008	7.84	3.06	0.19	0.36	0.35
zipcloak	3033	7.50	20.42	0.64	0.67	0.19
sim	3034	3.00	8.24	0.35	0.31	0.20
cdecl	3196	4.60	21.08	0.33	0.34	0.25
diff	3300	4.36	11.17	0.50	0.32	0.21
unzip	3416	9.88	10.57	0.40	0.27	0.29
assembler	3601	11.54	26.06	0.55	0.57	0.56
gnugo	3651	14.24	3.25	0.32	0.19	0.19
livc	4101	4.50	7.98	0.57	0.26	0.37
lharc	4250	4.62	12.72	0.79	0.34	0.36
patch	4608	6.94	18.80	0.51	0.35	0.40
simulator	5574	14.70	11.31	0.60	0.34	0.53
arc	5856	19.02	15.00	0.91	0.44	0.53
triangle	6119	8.42	8.88	0.64	0.72	0.31
tbl	6162	14.00	27.16	0.54	0.67	0.43
football	7313	10.22	9.86	0.83	0.37	0.46
flex	7376	11.46	116.82	0.74	0.82	0.67
zip	9288	18.12		1.80		1.01
072.sc	13690	16.52		2.05		2.08
spim	16740	25.38		2.23		3.78
larn	21184	37.78		2.36		8.03
tsl	27302	21.72		10.86		14.69
008.espresso	30510	40.72		6.17		6.09
moria	38572	45.84	77.89	7.02	3.02	52.07
TWMC	51627	152.22		11.06		9.47
nethack	58317	71.38		124.32		355.41

Table 3: Timing Data

C Raw Data

Table 3 shows the analysis times for the MOD$_C$ calculations broken into two passes, and for a simple compilation using Gnu's gcc compiler version 2.7.2 with no optimizations enabled. The numbers are as reported by the UNIX `time` utility, averaged over 5 executions on a *Sun Sparcstation 20*. The MOD$_C$(FS) and MOD$_C$(FI) times *do not* include the alias analysis times, but are simply the time taken to calculate the MOD$_C$ solution given the alias solution. The total analysis time is the sum of the two columns (columns 4 and 6 for MOD$_C$($FSAlias$) and columns 5 and 7 for MOD$_C$($FIAlias$)).

Tables 4 to 11 contain summary statistics for the MOD solution. These statistics are subdivided with respect to the type of fixed locations being modified. There are five types:

- **glo**: MOD information for global variables.
- **dyn**: MOD information for dynamic storage locations (heap storage creation sites).

	Average # Locations Modified					Average Percent of Worst Case					Maximum # Locations Modified				
program	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot
allroots	0.00	1.00	0.00	0.00	1.00	0%	100%	0%	0%	14%	0	1	0	0	1
fixoutput	0.00	1.80	0.00	0.00	1.80	0%	60%	0%	0%	15%	0	2	0	0	2
diffh	0.50	0.25	0.25	0.00	1.00	6%	25%	4%	0%	7%	1	1	1	0	1
travel	0.00	1.00	0.00	0.00	1.00	0%	33%	0%	0%	4%	0	1	0	0	1
ul	0.33	0.00	0.67	0.00	1.00	1%	0%	19%	0%	4%	1	0	1	0	1
plot2fig	0.62	0.25	0.00	0.25	1.12	3%	13%	0%	6%	4%	1	2	0	1	2
lex315	0.00	1.83	0.00	0.00	1.83	0%	61%	0%	0%	16%	0	2	0	0	2
compress	0.73	0.27	0.00	0.00	1.00	2%	27%	0%	0%	3%	1	1	0	0	1
clinpack	1.30	0.00	0.00	0.07	1.37	9%	0%	0%	7%	7%	3	0	0	1	3
loader	0.53	0.27	0.00	0.76	1.55	7%	4%	0%	17%	7%	1	2	0	9	9
mway	0.51	0.00	0.00	0.52	1.03	1%	0%	0%	14%	2%	1	0	0	2	2
ansitape	0.95	0.57	0.00	0.10	1.62	3%	14%	0%	10%	5%	7	2	0	1	7
stanford	0.64	0.19	0.00	0.24	1.07	2%	10%	0%	5%	3%	2	2	0	1	2
pokerd	0.61	0.32	0.00	0.34	1.27	6%	8%	0%	24%	6%	1	3	0	3	3
zipship	0.41	0.63	0.03	0.00	1.07	2%	21%	<1%	0%	4%	2	1	1	0	2
dixie	0.07	1.07	0.00	0.58	1.71	<1%	10%	0%	10%	4%	1	4	0	6	7
zipnote	0.22	0.93	0.00	0.01	1.16	2%	6%	0%	1%	3%	1	4	0	1	4
learn	1.00	0.12	0.02	0.08	1.22	2%	3%	<1%	5%	2%	2	3	1	2	3
xmodem	1.02	0.00	0.00	0.03	1.05	4%	0%	0%	2%	3%	2	0	0	1	2
compiler	1.00	0.00	0.00	0.00	1.00	3%	0%	0%	0%	3%	1	0	0	0	1
zipcloak	0.34	0.73	0.00	0.02	1.09	2%	6%	0%	2%	3%	1	2	0	1	2
sim	0.08	0.92	0.00	0.00	1.00	<1%	5%	0%	0%	1%	1	1	0	0	1
cdecl	0.40	0.84	0.04	0.00	1.28	1%	44%	<1%	0%	3%	1	2	1	0	2
diff	0.43	0.98	0.00	0.00	1.41	1%	8%	0%	0%	2%	2	3	0	0	3
unzip	0.83	0.54	0.04	0.23	1.63	1%	14%	<1%	15%	2%	5	2	1	3	7
assembler	0.55	0.22	0.00	0.74	1.51	3%	1%	0%	16%	3%	2	2	0	9	9
gnugo	0.11	0.00	0.00	1.07	1.18	1%	0%	0%	33%	4%	1	0	0	2	2
livc	1.32	0.00	0.00	0.00	1.32	2%	0%	0%	0%	2%	54	0	0	0	54
lharc	0.42	0.06	0.01	1.36	1.85	1%	2%	<1%	44%	3%	1	2	1	4	4
patch	0.39	0.81	0.03	0.00	1.23	1%	8%	<1%	0%	1%	1	2	1	0	2
simulator	0.14	0.42	0.00	1.44	2.00	1%	11%	0%	40%	6%	1	2	0	13	13
arc	0.56	0.31	0.01	0.81	1.69	1%	2%	<1%	29%	1%	3	2	1	8	8
triangle	0.07	0.27	0.00	0.67	1.01	7%	2%	0%	14%	2%	1	1	0	2	2
tbl	0.97	0.09	0.00	0.05	1.12	2%	5%	0%	1%	2%	8	1	1	4	8
football	1.04	0.00	0.00	0.00	1.05	2%	0%	0%	<1%	2%	3	0	0	1	3
flex	0.43	1.23	0.01	0.12	1.79	<1%	2%	<1%	5%	1%	3	7	1	2	7
moria	0.86	0.00	0.00	0.62	1.48	1%	<1%	0%	12%	2%	3	1	0	33	33

Table 4: Statistics for $\mathrm{MOD}_C(FSAlias)$ (through-dereference assignments)

- **loc**: MOD information for local variables of the enclosing procedure (including formal parameters).
- **nv**: (*non_visible*) MOD information for local variables of *other* procedures or of an earlier recursive instantiation of the enclosing procedure. Section 2.2 discusses *non_visible*s in more detail.
- **tot**: MOD information for all fixed locations.

There are three different summary statistics. **Average # Location Modified** (**Maximum # Location Modified**) is the average (maximum) number of fixed locations modified by statements (possibly procedures) of the type indicated by the tables. **Average Percent of Worst Case** is more complicated. The number of fixed locations potentially modified by an assignment is the sum of

- the number of globals in the program,
- the number of dynamic allocation sites,
- the number of locals in the enclosing procedure, and
- the number of locals of other procedures (including locals of earlier recursive instantiations of this procedure) accessible through globals and formals at the entry of the enclosing procedure.

	Average # Locations Modified					Average Percent of Worst Case					Maximum # Locations Modified				
program	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot
allroots	0.43	0.06	0.51	0.00	1.00	22%	6%	16%	0%	16%	1	1	1	0	1
fixoutput	0.83	0.10	0.11	0.00	1.03	10%	3%	3%	0%	7%	1	2	1	0	2
diffh	0.31	0.03	0.66	0.00	1.00	4%	3%	18%	0%	7%	1	1	1	0	1
travel	0.26	0.04	0.70	0.00	1.00	2%	1%	14%	0%	4%	1	1	1	0	1
ul	0.63	0.00	0.37	0.00	1.00	2%	0%	18%	0%	4%	1	0	1	0	1
plot2fig	0.50	0.04	0.45	0.03	1.01	2%	2%	13%	1%	4%	1	2	1	1	2
lex315	0.42	0.10	0.51	0.00	1.04	6%	3%	6%	0%	7%	1	2	1	0	2
compress	0.59	0.01	0.40	0.00	1.00	2%	2%	10%	0%	3%	1	1	1	0	1
clinpack	0.48	0.00	0.56	0.01	1.04	3%	0%	13%	1%	5%	3	0	1	1	3
loader	0.34	0.12	0.48	0.24	1.18	4%	2%	11%	5%	5%	1	2	1	9	9
mway	0.35	0.00	0.56	0.09	1.00	1%	0%	8%	3%	2%	1	0	1	2	2
ansitape	0.65	0.06	0.33	0.01	1.05	2%	2%	18%	1%	3%	7	2	1	1	7
stanford	0.35	0.03	0.60	0.03	1.01	1%	1%	25%	1%	3%	2	2	1	1	2
pokerd	0.25	0.08	0.66	0.07	1.05	3%	2%	15%	5%	5%	1	3	1	3	3
zipship	0.41	0.12	0.48	0.00	1.01	2%	4%	10%	0%	4%	2	1	1	0	2
dixie	0.25	0.23	0.55	0.11	1.13	1%	2%	15%	2%	3%	1	4	1	6	7
zipnote	0.43	0.27	0.33	0.00	1.04	3%	2%	8%	<1%	3%	1	4	1	1	4
learn	0.62	0.03	0.37	0.01	1.03	1%	1%	9%	1%	2%	2	3	1	2	3
xmodem	0.49	0.00	0.51	0.01	1.01	2%	0%	8%	<1%	3%	2	0	1	1	2
compiler	0.74	0.00	0.26	0.00	1.00	2%	0%	13%	0%	3%	1	0	1	0	1
zipcloak	0.48	0.21	0.33	0.00	1.02	3%	2%	9%	1%	3%	1	2	1	1	2
sim	0.21	0.17	0.62	0.00	1.00	1%	1%	5%	0%	1%	1	1	1	0	1
cdecl	0.71	0.05	0.25	0.00	1.02	2%	3%	6%	0%	2%	1	2	1	0	2
diff	0.40	0.19	0.48	0.00	1.07	1%	2%	12%	0%	2%	2	3	1	0	3
unzip	0.45	0.04	0.53	0.02	1.05	1%	1%	9%	1%	2%	5	2	1	3	7
assembler	0.36	0.13	0.41	0.32	1.22	2%	1%	13%	7%	3%	2	2	1	9	9
gnugo	0.50	0.00	0.35	0.18	1.03	3%	0%	9%	6%	4%	1	0	1	2	2
livc	0.51	0.00	0.58	0.00	1.09	1%	0%	22%	0%	1%	54	0	1	0	54
lharc	0.40	0.01	0.50	0.21	1.13	1%	<1%	15%	7%	2%	1	2	1	4	4
patch	0.49	0.16	0.39	0.00	1.04	1%	2%	8%	0%	1%	1	2	1	0	2
simulator	0.46	0.07	0.39	0.23	1.16	2%	2%	15%	6%	4%	1	2	1	13	13
arc	0.48	0.06	0.44	0.12	1.10	<1%	<1%	14%	4%	1%	3	2	1	8	8
triangle	0.04	0.07	0.74	0.15	1.00	4%	1%	5%	3%	2%	1	1	1	2	2
tbl	0.50	0.03	0.49	0.02	1.04	1%	2%	12%	<1%	1%	8	1	1	4	8
football	0.51	0.00	0.50	0.00	1.01	1%	0%	11%	<1%	2%	3	0	1	1	3
flex	0.53	0.25	0.32	0.02	1.13	<1%	<1%	7%	1%	1%	3	7	1	2	7
moria	0.35	0.00	0.62	0.16	1.12	<1%	0%	12%	3%	1%	3	1	1	33	33

Table 5: Statistics for $\mathrm{MOD}_C(FSAlias)$ (assignments)

	Average # Locations Modified					Average Percent of Worst Case					Maximum # Locations Modified				
program	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot
allroots	0.95	0.25	0.00	0.00	1.20	48%	25%	0%	0%	24%	2	1	0	0	3
fixoutput	5.46	2.69	0.00	0.00	8.15	68%	90%	0%	0%	64%	8	3	0	0	11
diffh	2.48	0.48	0.00	0.00	2.96	28%	48%	0%	0%	22%	9	1	0	0	10
travel	4.62	1.33	0.00	0.00	5.96	26%	44%	0%	0%	26%	18	3	0	0	21
ul	4.14	0.00	0.00	0.00	4.14	16%	0%	0%	0%	15%	26	0	0	0	26
plot2fig	2.82	0.23	0.05	0.00	3.10	13%	12%	1%	0%	11%	21	2	4	0	23
lex315	3.04	1.75	0.00	0.00	4.79	43%	58%	0%	0%	48%	7	3	0	0	10
compress	6.21	0.07	0.00	0.00	6.28	21%	7%	0%	0%	18%	30	1	0	0	31
clinpack	1.64	0.00	0.11	0.00	1.75	11%	0%	1%	0%	8%	15	0	1	0	15
loader	1.05	0.50	0.68	0.06	2.29	13%	7%	14%	4%	11%	8	7	3	1	15
mway	3.21	0.00	0.51	0.02	3.74	8%	0%	2%	2%	8%	39	0	8	1	39
ansitape	3.35	0.30	0.04	0.00	3.69	12%	8%	2%	0%	11%	29	4	1	0	33
stanford	2.66	0.17	0.06	0.06	2.96	9%	9%	1%	1%	9%	29	2	5	5	31
pokerd	0.84	0.22	0.16	0.00	1.22	8%	6%	2%	0%	7%	10	4	1	0	14
zipship	2.70	0.49	0.00	0.00	3.19	14%	16%	0%	0%	12%	19	3	0	0	22
dixie	5.10	1.83	0.31	0.00	7.24	23%	17%	4%	0%	18%	22	11	1	0	33
zipnote	1.37	1.34	0.03	0.00	2.73	9%	9%	<1%	0%	7%	13	15	1	0	28
learn	3.67	0.19	0.10	0.00	3.96	8%	5%	1%	0%	8%	46	4	2	0	50
xmodem	1.67	0.00	0.03	0.00	1.70	6%	0%	<1%	0%	5%	27	0	2	0	27
compiler	11.03	0.00	0.00	0.00	11.03	33%	0%	0%	0%	32%	33	0	0	0	33
zipcloak	1.31	1.00	0.01	0.00	2.32	8%	8%	<1%	0%	7%	16	12	1	0	28
sim	9.17	4.83	0.00	0.00	14.00	20%	24%	0%	0%	19%	45	20	0	0	65
cdecl	2.58	0.88	0.00	0.00	3.46	7%	44%	0%	0%	7%	36	2	0	0	38
diff	2.99	0.76	0.00	0.00	3.75	6%	6%	0%	0%	6%	52	12	0	0	64
unzip	7.55	1.26	0.09	0.02	8.92	13%	32%	2%	1%	13%	60	4	3	1	64
assembler	2.25	1.44	0.32	0.20	4.21	11%	9%	6%	7%	10%	20	16	5	5	36
gnugo	2.54	0.00	0.51	0.39	3.44	13%	0%	9%	13%	13%	20	0	3	3	20
livc	2.64	0.00	0.00	0.00	2.64	3%	0%	0%	0%	3%	78	0	0	0	78
lharc	3.79	0.17	0.13	0.07	4.17	7%	6%	3%	3%	7%	54	3	2	4	57
patch	2.73	0.68	0.00	0.00	3.41	4%	6%	0%	0%	4%	76	11	0	0	87
simulator	1.58	0.25	0.23	0.06	2.13	8%	6%	7%	5%	8%	20	4	2	3	24
arc	8.27	0.45	0.10	0.02	8.85	8%	3%	3%	2%	7%	106	16	1	1	122
triangle	0.16	0.98	0.91	0.16	2.21	16%	8%	3%	2%	8%	1	13	6	3	14
tbl	2.48	0.09	0.03	0.01	2.61	4%	5%	1%	<1%	4%	66	2	1	3	68
football	2.71	0.00	0.00	0.00	2.71	6%	0%	<1%	0%	5%	45	0	1	0	45
flex	7.74	6.32	0.06	0.01	14.12	5%	8%	1%	<1%	6%	153	75	3	1	228
moria	4.78	0.00	0.12	0.02	4.92	6%	<1%	2%	<1%	5%	86	1	4	22	87

Table 6: Statistics for $\mathrm{MOD}_C(FSAlias)$ (calls)

	Average # Locations Modified					Average Percent of Worst Case					Maximum # Locations Modified				
program	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot
allroots	1.62	0.62	2.38	0.00	4.62	81%	63%	75%	0%	85%	2	1	6	0	8
fixoutput	6.14	2.71	1.00	0.00	9.86	77%	91%	29%	0%	82%	8	3	4	0	15
diffh	3.40	0.53	1.93	0.00	5.87	38%	53%	53%	0%	47%	9	1	7	0	15
travel	6.62	1.50	2.56	0.00	10.69	37%	50%	63%	0%	46%	18	3	12	0	21
ul	8.80	0.00	1.33	0.00	10.13	34%	0%	53%	0%	37%	26	0	4	0	30
plot2fig	4.96	0.37	1.04	0.15	6.52	24%	19%	30%	100%	25%	21	2	9	4	30
lex315	3.61	1.83	1.67	0.00	7.11	52%	61%	28%	0%	61%	7	3	16	0	17
compress	9.12	0.19	2.50	0.00	11.81	30%	19%	63%	0%	34%	30	1	9	0	40
clinpack	4.36	0.00	3.36	0.07	7.79	29%	0%	79%	100%	41%	15	0	12	1	27
loader	2.03	1.26	2.65	1.10	7.03	25%	18%	65%	82%	32%	8	7	18	9	33
mway	7.59	0.00	4.91	0.86	13.36	19%	0%	68%	57%	28%	40	0	40	8	52
ansitape	6.39	0.86	1.44	0.08	8.78	22%	22%	72%	60%	25%	29	4	5	1	34
stanford	4.10	0.25	1.58	0.10	6.04	14%	13%	75%	100%	19%	29	2	8	5	31
pokerd	2.37	0.78	3.19	0.37	6.70	24%	19%	82%	64%	36%	10	4	12	3	21
zipship	6.29	1.21	2.79	0.00	10.29	33%	41%	79%	0%	40%	19	3	11	0	30
dixie	5.94	2.56	2.89	0.33	11.72	27%	23%	78%	14%	31%	22	11	12	6	45
zipnote	3.05	3.40	2.20	0.05	8.70	20%	23%	65%	100%	25%	15	15	12	1	39
learn	7.08	0.47	2.67	0.19	10.42	15%	12%	69%	41%	20%	46	4	13	2	50
xmodem	4.75	0.00	3.04	0.11	7.89	18%	0%	54%	67%	24%	27	0	21	2	32
compiler	18.82	0.00	1.10	0.00	19.92	57%	0%	64%	0%	58%	33	0	4	0	35
zipcloak	2.70	1.90	2.00	0.03	6.63	16%	16%	63%	20%	20%	17	12	12	1	39
sim	13.12	6.65	6.71	0.00	26.47	29%	33%	71%	0%	35%	45	20	20	0	85
cdecl	8.85	0.67	1.48	0.00	11.00	23%	33%	42%	0%	25%	38	2	18	0	50
diff	6.65	1.95	2.86	0.00	11.47	13%	16%	77%	0%	17%	52	12	11	0	67
unzip	12.62	1.48	4.00	0.20	18.30	21%	37%	70%	100%	27%	60	4	15	3	71
assembler	5.32	3.74	2.85	1.68	13.58	27%	23%	74%	69%	32%	20	16	13	15	47
gnugo	4.48	0.00	2.93	1.00	8.41	22%	0%	72%	92%	34%	20	0	9	4	26
livc	5.87	0.00	1.34	0.00	7.22	8%	0%	58%	0%	9%	78	0	13	0	81
lharc	6.29	0.29	2.51	0.37	9.45	12%	10%	69%	36%	15%	54	3	11	4	58
patch	8.66	1.75	2.27	0.00	12.68	11%	16%	48%	0%	14%	76	11	20	0	95
simulator	3.21	0.71	1.94	0.84	6.70	16%	18%	82%	89%	24%	20	4	7	24	29
arc	13.57	1.07	2.26	0.22	17.12	12%	7%	72%	52%	13%	110	16	10	8	126
triangle	0.42	2.89	13.21	2.26	18.79	42%	22%	90%	43%	55%	1	13	41	6	55
tbl	6.99	0.29	2.06	0.08	9.42	11%	15%	62%	29%	13%	66	2	15	4	68
football	5.17	0.00	3.03	0.02	8.22	12%	0%	71%	20%	16%	45	0	36	1	63
flex	15.58	9.64	2.55	0.16	27.93	10%	13%	62%	38%	12%	153	75	22	3	228
moria	7.88	0.01	3.80	0.87	12.56	9%	1%	75%	69%	13%	86	1	24	66	87

Table 7: Statistics for $\mathrm{MOD}_C(FSAlias)$ (procedures)

	Average # Locations Modified					Average Percent of Worst Case					Maximum # Locations Modified				
program	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot
allroots	1.00	1.00	0.00	0.00	2.00	33%	100%	0%	0%	25%	1	1	0	0	2
fixoutput	0.00	1.80	0.00	0.00	1.80	0%	60%	0%	0%	15%	0	2	0	0	2
diffh	1.00	0.50	0.25	0.25	2.00	11%	50%	4%	25%	13%	1	1	1	1	3
travel	1.00	3.00	0.33	1.67	6.00	6%	100%	3%	47%	19%	1	3	1	2	6
ul	0.33	0.00	0.67	0.00	1.00	1%	0%	19%	0%	3%	1	0	1	0	1
plot2fig	0.75	0.25	0.00	0.25	1.25	3%	13%	0%	6%	4%	1	2	0	1	3
lex315	0.00	1.83	0.00	0.00	1.83	0%	61%	0%	0%	16%	0	2	0	0	2
compress	1.64	0.27	0.00	0.00	1.91	6%	27%	0%	0%	5%	3	1	0	0	3
clinpack	2.47	0.00	0.00	0.07	2.53	16%	0%	0%	7%	13%	3	0	0	1	3
loader	0.56	0.27	0.03	1.23	2.09	7%	4%	1%	4%	5%	1	2	1	12	13
mway	0.54	0.00	0.00	0.55	1.08	1%	0%	0%	4%	2%	1	0	0	2	2
ansitape	2.62	0.95	0.00	0.86	4.43	9%	24%	0%	21%	11%	8	2	0	3	13
stanford	0.64	0.33	0.00	0.24	1.21	2%	17%	0%	5%	3%	2	2	0	1	2
pokerd	0.61	0.56	0.00	1.49	2.66	6%	14%	0%	15%	10%	1	3	0	7	8
zipship	0.80	0.85	0.03	0.00	1.68	4%	28%	<1%	0%	6%	4	2	1	0	6
dixie	0.07	2.79	0.18	4.59	7.63	<1%	25%	5%	34%	15%	1	4	1	6	9
zipnote	0.43	1.84	0.08	0.08	2.43	3%	12%	1%	2%	6%	1	5	1	1	6
learn	1.36	0.17	0.14	1.12	2.78	3%	4%	2%	10%	4%	3	3	2	6	9
xmodem	1.20	0.00	0.00	0.03	1.23	4%	0%	0%	1%	3%	3	0	0	1	3
compiler	1.00	0.00	0.00	0.00	1.00	3%	0%	0%	0%	3%	1	0	0	0	1
zipcloak	0.49	1.12	0.04	0.02	1.67	3%	9%	<1%	1%	4%	1	4	1	1	5
sim	0.12	0.96	0.00	0.00	1.08	<1%	5%	0%	0%	1%	2	2	0	0	2
cdecl	3.04	1.04	0.08	0.52	4.68	8%	54%	1%	33%	11%	5	2	1	1	8
diff	0.44	2.09	0.00	0.00	2.53	1%	17%	0%	0%	4%	2	4	0	0	4
unzip	2.83	1.38	0.04	0.23	4.48	5%	35%	<1%	3%	6%	5	3	1	3	8
assembler	1.03	1.38	0.21	3.65	6.26	5%	9%	2%	8%	7%	3	7	4	17	27
gnugo	0.11	0.00	0.14	1.09	1.34	1%	0%	3%	6%	3%	1	0	1	2	2
livc	1.63	0.00	0.00	0.00	1.63	2%	0%	0%	0%	2%	77	0	0	0	77
lharc	0.67	0.14	0.01	1.76	2.59	1%	5%	<1%	6%	3%	3	2	1	6	8
patch	2.61	1.80	0.04	0.53	4.98	3%	17%	1%	20%	5%	8	4	1	2	14
simulator	0.14	0.42	0.02	2.62	3.20	1%	11%	1%	4%	3%	1	2	1	14	14
arc	0.78	0.41	0.02	1.45	2.66	1%	3%	<1%	7%	2%	6	2	1	9	10
triangle	0.07	0.31	0.00	1.30	1.68	7%	2%	0%	3%	2%	1	2	0	8	8
tbl	1.15	0.18	0.04	0.29	1.66	2%	9%	1%	3%	2%	8	2	1	4	8
football	1.18	0.00	0.00	0.04	1.22	3%	0%	0%	1%	2%	9	0	0	2	9
flex	0.61	1.67	0.02	0.33	2.63	<1%	2%	<1%	2%	1%	3	7	1	5	13
zip	0.86	4.83	0.02	0.02	5.74	1%	12%	<1%	<1%	4%	5	15	1	2	17
072.sc	6.53	0.65	0.04	3.21	10.44	10%	22%	1%	24%	12%	10	1	1	5	16
spim	6.59	8.41	0.01	0.26	15.27	5%	57%	<1%	1%	8%	8	10	1	13	18
larn	10.71	2.17	0.01	2.82	15.72	7%	23%	<1%	8%	8%	20	4	1	13	28
tsl	1.77	31.54	0.00	0.99	34.30	6%	85%	0%	3%	35%	3	36	0	4	39
008.espresso	0.53	10.11	0.01	1.65	12.31	1%	6%	<1%	3%	4%	11	21	2	7	25
moria	14.13	0.00	0.22	82.79	97.14	14%	<1%	3%	20%	18%	25	1	4	149	174
TWMC	0.30	6.11	0.00	0.05	6.46	<1%	3%	0%	<1%	1%	18	33	1	32	33
nethack	60.46	23.21	0.12	45.02	128.81	24%	50%	2%	47%	32%	71	27	4	53	151

Table 8: Statistics for $\text{MOD}_C(FIAlias)$ (through-dereference assignments)

Percent of Worst Case is simply the number of fixed locations modified divided by the number of potentially modified locations. The **Average Percent of Worst Case** is the average of Percent of Worst Case over all assignments, calls, or procedures depending on the statement kind. For some statements/procedures, the number of possible locals and the number of *non_visible*s are zero. In these cases, "0%" is used as the **percent/assign**.

	Average # Locations Modified					Average Percent of Worst Case					Maximum # Locations Modified				
program	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot
allroots	0.47	0.06	0.51	0.00	1.04	16%	6%	16%	0%	14%	1	1	1	0	2
fixoutput	0.83	0.10	0.11	0.00	1.03	10%	3%	3%	0%	7%	1	2	1	0	2
diffh	0.34	0.04	0.66	0.01	1.05	4%	4%	18%	1%	7%	1	1	1	1	3
travel	0.28	0.07	0.71	0.03	1.09	2%	2%	14%	1%	4%	1	3	1	2	6
ul	0.63	0.00	0.37	0.00	1.00	2%	0%	18%	0%	3%	1	0	1	0	1
plot2fig	0.51	0.04	0.45	0.03	1.03	2%	2%	13%	1%	3%	1	2	1	1	3
lex315	0.42	0.10	0.51	0.00	1.04	6%	3%	6%	0%	7%	1	2	1	0	2
compress	0.62	0.01	0.40	0.00	1.04	2%	2%	10%	0%	3%	3	1	1	0	3
clinpack	0.61	0.00	0.56	0.01	1.17	4%	0%	13%	1%	5%	3	0	1	1	3
loader	0.35	0.12	0.49	0.40	1.35	4%	2%	11%	1%	3%	1	2	1	12	13
mway	0.36	0.00	0.56	0.10	1.01	1%	0%	8%	1%	2%	1	0	1	2	2
ansitape	0.78	0.09	0.33	0.07	1.26	3%	2%	18%	2%	3%	8	2	1	3	13
stanford	0.35	0.04	0.60	0.03	1.02	1%	2%	25%	1%	3%	2	2	1	1	2
pokerd	0.25	0.12	0.66	0.30	1.33	3%	3%	15%	3%	5%	1	3	1	7	8
zipship	0.48	0.16	0.48	0.00	1.12	3%	5%	10%	0%	4%	4	2	1	0	6
dixie	0.25	0.55	0.58	0.85	2.23	1%	5%	16%	6%	4%	1	4	1	6	9
zipnote	0.49	0.50	0.35	0.02	1.35	3%	3%	9%	1%	3%	1	5	1	1	6
learn	0.67	0.03	0.38	0.15	1.24	1%	1%	9%	1%	2%	3	3	2	6	9
xmodem	0.53	0.00	0.51	0.01	1.05	2%	0%	8%	<1%	3%	3	0	1	1	3
compiler	0.74	0.00	0.26	0.00	1.00	2%	0%	13%	0%	3%	1	0	1	0	1
zipcloak	0.52	0.30	0.34	0.00	1.17	3%	3%	9%	<1%	3%	1	4	1	1	5
sim	0.21	0.18	0.62	0.00	1.01	1%	1%	5%	0%	1%	2	2	1	0	2
cdecl	0.86	0.06	0.25	0.03	1.21	2%	3%	6%	2%	2%	5	2	1	1	8
diff	0.40	0.39	0.48	0.00	1.27	1%	3%	12%	0%	2%	2	4	1	0	4
unzip	0.59	0.10	0.53	0.02	1.25	1%	3%	9%	<1%	2%	5	3	1	3	8
assembler	0.57	0.63	0.50	1.59	3.30	3%	4%	14%	3%	4%	3	7	4	17	27
gnugo	0.50	0.00	0.37	0.18	1.06	3%	0%	10%	1%	2%	1	0	1	2	2
livc	0.59	0.00	0.58	0.00	1.17	1%	0%	22%	0%	1%	77	0	1	0	77
lharc	0.44	0.03	0.50	0.27	1.25	1%	1%	15%	1%	1%	3	2	1	6	8
patch	0.89	0.34	0.40	0.09	1.72	1%	3%	8%	4%	2%	8	4	1	2	14
simulator	0.46	0.07	0.39	0.42	1.35	2%	2%	15%	1%	1%	1	2	1	14	14
arc	0.51	0.07	0.44	0.21	1.24	1%	1%	14%	1%	1%	6	2	1	9	10
triangle	0.04	0.08	0.74	0.29	1.15	4%	1%	5%	1%	2%	1	2	1	8	8
tbl	0.56	0.06	0.50	0.09	1.20	1%	3%	12%	1%	2%	8	2	1	4	8
football	0.54	0.00	0.50	0.01	1.06	1%	0%	11%	<1%	2%	9	0	1	2	9
flex	0.56	0.32	0.32	0.06	1.27	<1%	<1%	7%	<1%	1%	3	7	1	5	13
zip	0.47	1.06	0.48	0.00	2.01	1%	3%	11%	0%	1%	5	15	1	2	17
072.sc	1.08	0.07	0.53	0.35	2.04	2%	2%	13%	3%	2%	10	1	1	5	16
spim	1.98	2.02	0.35	0.06	4.41	2%	14%	9%	<1%	2%	8	10	1	13	18
larn	1.11	0.14	0.49	0.18	1.92	1%	1%	17%	1%	1%	20	4	1	13	28
tsl	0.49	7.90	0.69	0.25	9.32	2%	21%	21%	1%	9%	3	36	1	4	39
008.espresso	0.25	3.08	0.58	0.50	4.41	<1%	2%	10%	1%	2%	11	21	2	7	25
moria	3.71	0.00	0.67	20.98	25.36	4%	0%	12%	5%	5%	25	1	4	149	174
TWMC	0.17	2.28	0.54	0.02	3.02	<1%	1%	4%	0%	1%	18	33	1	32	33
nethack	13.04	4.90	0.51	9.49	27.95	5%	10%	14%	10%	7%	71	27	4	53	151

Table 9: Statistics for MOD$_C$($FIAlias$) (all assignments including through-dereferences)

	Average # Locations Modified					Average Percent of Worst Case					Maximum # Locations Modified				
program	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot
allroots	1.20	0.25	0.00	0.00	1.45	40%	25%	0%	0%	25%	3	1	0	0	4
fixoutput	5.46	2.69	0.00	0.00	8.15	68%	90%	0%	0%	64%	8	3	0	0	11
diffh	2.96	0.52	0.00	0.52	4.00	33%	52%	0%	52%	27%	9	1	0	1	11
travel	4.62	2.12	0.21	1.92	8.88	26%	71%	3%	43%	32%	18	3	1	3	24
ul	4.14	0.00	0.00	0.67	4.81	16%	0%	0%	33%	16%	26	0	0	2	28
plot2fig	2.94	0.23	0.05	0.26	3.47	13%	12%	1%	6%	11%	22	2	4	4	28
lex315	3.04	1.75	0.00	0.00	4.79	43%	58%	0%	0%	48%	7	3	0	0	10
compress	6.79	0.07	0.00	0.14	7.00	23%	7%	0%	7%	19%	30	1	0	2	33
clinpack	2.25	0.00	0.11	0.03	2.39	15%	0%	1%	3%	10%	15	0	1	1	16
loader	1.60	0.50	1.27	6.76	10.14	20%	7%	25%	26%	21%	8	7	5	29	44
mway	3.33	0.00	0.56	1.37	5.26	8%	0%	2%	11%	9%	39	0	8	15	54
ansitape	6.86	1.46	0.15	1.89	10.37	23%	37%	8%	51%	26%	30	4	1	3	37
stanford	2.71	0.20	0.06	0.31	3.29	9%	10%	1%	6%	9%	29	2	5	5	36
pokerd	0.84	0.43	0.19	2.02	3.48	8%	11%	3%	24%	12%	10	4	1	10	24
zipship	3.28	0.72	0.00	0.11	4.11	17%	24%	0%	8%	14%	19	3	0	2	24
dixie	5.34	3.18	1.83	5.14	15.49	24%	29%	17%	53%	30%	22	11	6	14	47
zipnote	1.75	2.34	0.04	0.07	4.20	12%	16%	<1%	2%	10%	13	15	1	2	30
learn	4.15	0.19	0.19	1.44	5.96	9%	5%	2%	13%	9%	47	4	2	9	60
xmodem	1.76	0.00	0.03	0.10	1.89	7%	0%	<1%	3%	5%	27	0	2	4	31
compiler	11.03	0.00	0.02	0.10	11.16	33%	0%	1%	10%	31%	33	0	1	1	34
zipcloak	1.58	1.57	0.01	0.05	3.22	9%	13%	<1%	1%	8%	16	12	1	2	30
sim	9.17	5.00	0.00	0.07	14.24	20%	25%	0%	7%	19%	45	20	0	1	66
cdecl	5.75	1.32	0.49	0.18	7.73	14%	66%	3%	9%	14%	38	2	1	1	41
diff	3.05	1.06	0.00	0.12	4.23	6%	9%	0%	4%	6%	52	12	0	3	67
unzip	11.17	2.42	0.17	1.61	15.37	19%	61%	5%	21%	20%	60	4	3	8	72
assembler	2.93	2.65	0.58	6.16	12.32	15%	17%	8%	13%	14%	20	16	6	49	85
gnugo	2.54	0.00	0.53	1.93	5.00	13%	0%	10%	10%	12%	20	0	3	21	41
livc	2.82	0.00	0.00	0.03	2.85	3%	0%	0%	2%	3%	87	0	0	2	89
lharc	4.26	0.25	0.21	2.10	6.82	8%	8%	4%	8%	8%	54	3	5	23	80
patch	5.11	1.56	0.00	0.67	7.35	7%	14%	0%	23%	7%	78	11	0	3	92
simulator	1.60	0.25	0.29	10.99	13.13	8%	6%	9%	15%	13%	20	4	2	74	98
arc	8.58	0.49	0.13	1.58	10.79	8%	3%	3%	8%	7%	106	16	2	20	142
triangle	0.16	0.98	1.00	2.65	4.79	16%	8%	4%	9%	9%	1	13	6	40	54
tbl	2.50	0.15	0.03	0.47	3.16	4%	7%	1%	5%	4%	66	2	1	9	77
football	3.56	0.00	0.01	0.82	4.40	8%	0%	<1%	12%	8%	45	0	1	7	52
flex	9.71	7.86	0.13	3.12	20.82	6%	11%	1%	21%	8%	155	75	4	16	246
zip	9.57	11.19	0.04	0.21	21.01	10%	28%	1%	2%	13%	96	40	2	10	146
072.sc	10.40	0.84	0.08	3.59	14.91	15%	28%	1%	26%	16%	69	3	2	14	86
spim	15.51	6.41	0.35	4.09	26.36	12%	43%	1%	13%	14%	132	15	11	24	171
larn	18.00	2.95	0.06	3.49	24.50	11%	30%	1%	9%	11%	167	10	4	34	211
tsl	2.86	24.22	0.03	1.85	28.96	10%	64%	1%	6%	28%	30	38	4	30	98
008.espresso	2.11	16.67	0.12	3.13	22.02	3%	10%	1%	5%	7%	65	170	5	56	291
moria	9.02	0.00	0.55	58.40	67.97	9%	<1%	6%	14%	13%	100	1	9	415	516
TWMC	6.05	8.28	0.56	6.25	21.14	2%	3%	3%	8%	4%	249	247	13	81	577
nethack	54.28	18.00	0.17	30.83	103.29	22%	38%	4%	32%	26%	252	47	4	86	385

Table 10: Statistics for $\text{MOD}_C(FIAlias)$ (calls)

	Average # Locations Modified					Average Percent of Worst Case					Maximum # Locations Modified				
program	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot	glob	dyn	loc	nv	tot
allroots	2.25	0.62	2.38	0.00	5.25	75%	63%	75%	0%	82%	3	1	6	0	8
fixoutput	6.14	2.71	1.00	0.00	9.86	77%	91%	29%	0%	82%	8	3	4	0	15
diffh	3.80	0.60	1.93	0.53	6.87	42%	60%	53%	57%	51%	9	1	7	1	16
travel	6.62	2.06	2.56	1.88	13.12	37%	69%	63%	42%	47%	18	3	12	3	24
ul	8.80	0.00	1.33	0.93	11.07	34%	0%	53%	47%	38%	26	0	4	2	32
plot2fig	5.15	0.37	1.04	0.59	7.15	23%	19%	30%	15%	24%	22	2	9	4	35
lex315	3.61	1.83	1.67	0.00	7.11	52%	61%	28%	0%	61%	7	3	16	0	17
compress	9.56	0.19	2.50	0.25	12.50	32%	19%	63%	13%	35%	30	1	9	2	40
clinpack	4.79	0.00	3.36	0.21	8.36	32%	0%	79%	23%	42%	15	0	12	1	27
loader	2.45	1.26	2.65	9.19	15.55	31%	18%	65%	34%	33%	8	7	18	29	56
mway	7.73	0.00	4.91	3.41	16.05	19%	0%	68%	26%	26%	40	0	40	15	59
ansitape	8.83	1.75	1.44	2.00	14.03	29%	44%	72%	52%	36%	30	4	5	3	38
stanford	4.10	0.25	1.58	0.52	6.46	14%	13%	75%	11%	17%	29	2	8	5	36
pokerd	2.37	1.07	3.19	3.81	10.44	24%	27%	82%	41%	38%	10	4	12	10	29
zipship	7.07	1.50	2.79	0.36	11.71	37%	50%	79%	21%	43%	19	3	11	2	31
dixie	5.94	3.53	2.89	5.08	17.44	27%	32%	78%	38%	35%	22	11	12	14	53
zipnote	3.20	4.05	2.20	0.35	9.80	21%	27%	65%	10%	26%	15	15	12	2	40
learn	7.86	0.47	2.67	2.58	13.58	17%	12%	69%	23%	21%	47	4	13	9	60
xmodem	5.14	0.00	3.04	0.54	8.71	19%	0%	54%	13%	24%	27	0	21	4	36
compiler	18.82	0.00	1.10	0.38	20.31	57%	0%	64%	40%	58%	33	0	4	1	36
zipcloak	2.80	2.20	2.00	0.20	7.20	17%	18%	63%	5%	20%	17	12	12	2	40
sim	13.12	6.71	6.71	0.12	26.65	29%	34%	71%	13%	35%	45	20	20	1	85
cdecl	10.00	0.73	1.48	0.33	12.55	25%	36%	42%	18%	27%	40	2	18	1	52
diff	6.70	2.63	2.86	0.30	12.49	13%	22%	77%	10%	18%	52	12	11	3	70
unzip	15.07	2.20	4.00	2.15	23.43	25%	55%	70%	27%	31%	60	4	15	8	79
assembler	6.30	5.72	2.85	17.00	31.87	32%	36%	74%	36%	36%	20	16	13	49	94
gnugo	4.48	0.00	2.93	3.83	11.24	22%	0%	72%	20%	26%	20	0	9	21	44
livc	6.39	0.00	1.34	0.07	7.80	7%	0%	58%	3%	8%	87	0	13	2	92
lharc	6.89	0.41	2.51	3.46	13.26	13%	14%	69%	13%	15%	54	3	11	23	81
patch	11.71	2.75	2.27	0.95	17.68	15%	25%	48%	33%	18%	78	11	20	3	100
simulator	3.23	0.71	1.94	20.94	26.82	16%	18%	82%	29%	27%	20	4	7	74	102
arc	13.99	1.12	2.26	2.45	19.82	13%	7%	72%	12%	13%	110	16	10	20	146
triangle	0.42	2.89	13.21	8.26	24.79	42%	22%	90%	24%	38%	1	13	41	40	56
tbl	7.05	0.39	2.06	1.24	10.73	11%	19%	62%	14%	14%	66	2	15	9	77
football	6.02	0.00	3.03	0.97	10.02	13%	0%	71%	14%	17%	45	0	36	7	69
flex	17.55	11.06	2.55	3.55	34.70	11%	15%	62%	23%	14%	155	75	22	16	246
zip	8.19	8.19	2.57	0.45	19.40	9%	21%	71%	3%	13%	96	40	21	10	167
072.sc	10.96	0.73	3.01	3.32	18.02	16%	24%	78%	24%	20%	69	3	32	14	116
spim	17.72	6.84	1.57	4.30	30.43	13%	46%	39%	13%	17%	132	15	78	24	174
larn	28.81	3.74	1.72	6.38	40.65	17%	37%	62%	17%	19%	168	10	14	34	215
tsl	3.70	26.69	1.99	2.92	35.30	12%	70%	63%	10%	35%	30	38	14	30	100
008.espresso	3.93	24.29	4.58	5.45	38.25	6%	14%	88%	9%	13%	65	170	22	56	307
moria	15.41	0.01	3.81	96.08	115.31	15%	1%	75%	23%	22%	100	1	24	415	516
TWMC	10.68	14.60	9.93	7.49	42.70	4%	6%	90%	10%	7%	249	247	61	81	592
nethack	71.68	23.13	2.13	40.06	137.01	28%	49%	61%	42%	34%	253	47	32	86	393

Table 11: Statistics for $\text{MOD}_C(FIAlias)$ (procedures)