RETROSPECTIVE:

Interprocedural Side-Effect Analysis in Linear Time

Keith D. Cooper
Department of Computer Science
Rice University
Houston, TX 77005-1892
keith@rice.edu

Ken Kennedy
Department of Computer Science
Rice University
Houston, TX 77005-1892
ken@rice.edu

1. INTRODUCTION

Through the mid-1970s, compiler analysis and optimization was, for the most part, restricted to single procedures. This type of analysis was sufficient because optimizations focused on issues like register allocation and optimization of array indexing calculations and in Fortran, the most commonly optimized language of the time, procedure calls occurred infrequently, possibly because they were known to be expensive.

The confluence of two trends, toward architectures that employed multiprocessor parallelism and toward programming methodologies that used more procedures, changed this situation. To optimize for parallel hardware, compilers had to turn to more substantial transformations that involved restructuring whole loop nests. In increasingly many cases (because of improved programming methodologies) these loop nests included procedure and function calls. Hence, ways of reasoning about the side effects of those procedure and function calls were needed. In addition, if the procedure could be compiled with knowledge of the calling context, the running time of the procedure could be reduced substantially by specialization, particularly if the procedure were called from deep within a loop nest.

However, most compiler developers were cautious about using interprocedural analysis for two reasons. First, since the number of procedures in a program could become very large, no practical algorithm could afford to be highly non-linear in the size of the call graph. Second, if interprocedural information were relied on to perform optimizations in some procedure, the object code for that procedure would be a function of the source of the entire program rather than of the procedure in question.

The paper in this volume established that one important class of interprocedural analysis problems, the alias-free, flow-insensitive side-effect problems, could be solved in time proportional to the size of the call graph times the number of variables being tracked. A later paper showed how to extend this algorithm to solve the complete problem in the same time bound [8]. Because the solution time is proportional to the size of the solution, no faster bound is possible.

The second inhibitor to acceptance of interprocedural analysis and optimization was addressed by the introduction of "recompilation analysis" [9, 4], which achieved the illusion of separate compilation even when compiling in the presence of interprocedural analysis.

20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection, 2003.

2. BACKGROUND

Our interest in interprocedural analysis and optimization began with Keith Cooper's 1981 thesis proposal, which aimed to make it possible to extend languages by adding functionality from procedure and function libraries as intrinsic operations in the language, primarily through procedure specialization and inlining. This document laid out an ambitious program of research that, quite frankly, we are still working to complete even today.

At the time, we believed that we could not rely exclusively on systematic inlining to achieve the goal, simply because the compile times would become intolerably long. Thus some portion of the optimization strategy would require interprocedural analysis. There were a number of important early works on interprocedural analysis [1, 17], but the work reported in this paper was most deeply influenced by Barth [3] and Banning [2].

Barth formulated interprocedural side effect analysis as a collection of relations, demonstrating that the problem could be solved in using transitive closure in $O(N^3 + V^3)$ time, where N is the number of procedures and V is the number of variables in the program. Barth also separated the treatment of side effects to global variables and reference parameters, an idea that we would employ in our two later papers. Although Barth claimed that the solutions were practical, most compiler developers were reluctant to use an algorithm that could take time that was cubic in either the number of procedures or number of variables because either or both of these might extraordinarily large.

Banning defined the crucial distinction between flow-sensitive and flow-insensitive interprocedural analysis problems. He also formulated the solution of flow-insensitive problems as a data flow analysis problem on the call graph. Banning developed a clever partition of the flow-insensitive side effect problem into an alias-free "direct" side-effect problem and the problem of computing and integrating aliases into the solution. He suggested using standard data-flow algorithms for the solution, but did not provide a complexity analysis. Myers [15] later showed that flow-sensitive problems were intractable in the most general formulation. The question about how fast we could solve the flow-insensitive problems remained open.

In the work leading to Cooper's dissertation, we were trying to determine the complexity of Banning's approach and we observed that the data flow problem resulting from his formulation was neither "rapid" [13] nor "fast" [10], which meant that neither the iterative algorithm nor the best elimination methods could be used to achieve a fast time running time. Just prior Cooper's thesis defense, we discovered that we could get around the problem if we could bound the number of parameters to any procedure in the program by some (perhaps large) constant. We then built on Barth's

work by decomposing the problem into two subproblems: determining side effects to global variables and determining side effects to reference parameters. This led to our 1984 PLDI paper [7] which used an old result of Tarjan's on solution of path problems [18] to achieve a near-linear time bound.

The final insight came later, when we discovered that we could formulate the two subproblems as reachability algorithms on different graphs—the call graph for side effects to globals and the *parameter binding graph* for reference parameters. As a result, each of these subproblems could be solved in no more than $O((N+E)V)$ time, where N is the number of procedures in the program, E is the number of call sites and V is the number of variables in the program. Since the solution to the problem has the same size, this algorithm is effectively optimal. Furthermore, if you could do set operations of $O(V)$ length in constant time using bit vector operations, the algorithm would be linear in the size of the call graph. This is central result of the paper included in this volume.

There remained the issue of whether the alias information could be computed and integrated into the solution to produce precise side-effect information in the same time bound. This was resolved positively by our 1989 POPL paper [8]. The combined result is an algorithm for flow-insensitive side effect analysis that is as fast as possible. The algorithm works for any standard lexically-scoped language with recursion.

Of course all of this requires the construction of a call graph. The best known procedure for constructing a precise call graph for Fortran in the presence of procedure parameters was Ryder's algorithm [16], which was proved to work for recursive procedures without change by Callahan et al [5]. In 1992, Kennedy and Hall produced an effectively linear algorithm that sacrificed some precision for speed [11].

3. INFLUENCE

The paper included in this volume resolved the issue of complexity of flow-insensitive side-effect analysis for scalar variable by presenting an algorithm that was both easy to understand and effectively optimal. In order to support parallelization, however, it was critical to extend side-effect analysis to sections of arrays. Interestingly, if the array sections form a lattice with the finite descending chain property, the algorithm presented here can be adapted to solve the side effect problem for those sections in the same asymptotic time bound [6, 12] (actually a constant time slower where the constant is proportional to the depth of the lattice).

Of course, the interprocedural problems that people really want to solve are the flow-sensitive ones. It is much more important to know that there is an upwards exposed use down a call chain than it is to know there is a use, whether upwards exposed or not. Needless to say, Myers' negative result inhibited progress in this arena. Nevertheless, many interprocedural compilers include flow-sensitive analyses, with measures to reduce accuracy if running times become overly long. We believe that this is sound approach, because most problems can be solved in acceptable running times using the iterative algorithm with dynamic choice of procedures to visit.

Although there have not been many commercial compilers that systematically perform interprocedural analysis, the Convex Application Compiler was notable because it was among the first [14]. A number of later compilers incorporated interprocedural side effect analysis, including compilers from Cray and Digital (now HP). Most of these use some variant of the algorithm from presented in this paper.

REFERENCES

[1] F. E. Allen. Interprocedural data flow analysis. In *Proceedings of the IFIP Congress 1974*, pages 398–402, Amsterdam, 1974. North Holland.

[2] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the Sixth Annual Symposium on Principles of Programming Languages*. ACM, Jan. 1979.

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

[4] M. Burke and L. Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[5] D. Callahan, A. Carle, M. W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, Apr. 1990.

[6] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[7] K. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the SIGPLAN 84 Sym. on Compiler Construction*. ACM, June 1984.

[8] K. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 49–59. ACM, January 1989.

[9] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization on the design of a software development environment. In *Proceedings of the SIGPLAN 85 Symposium on Compiler Construction*, June 1985.

[10] S. L. Graham and M. Wegman. A fast and usually linear algorithm for global data flow analysis. *Journal of the ACM*, 23(1):172–202, 1976.

[11] M. W. Hall and K. Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3), Sept. 1992.

[12] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), July 1991.

[13] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, Jan. 1976.

[14] R. Metzger and P. Smith. The CONVEX application compiler. *Fortran Journal*, 3(1):8–10, 1991.

[15] E. Myers. A precise inter-procedural data flow algorithm. In *Conference Record of the Eighth Annual Symposium on Principles of Programming Languages*. ACM, Jan. 1981.

[16] B. Ryder. Constructing the call graph of a program. *IEEE Trans. on Software Engineering*, SE-5:216–225, may 1979.

[17] T. C. Spillman. Exposing side-effects in a PL/I optimizing compiler. In *Proceedings of the IFIP Congress 1971*, pages 376–381, Amsterdam, 1971. North Holland.

[18] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

Interprocedural Side-Effect Analysis in Linear Time

Keith D. Cooper
Ken Kennedy

Department of Computer Science†
Rice University
Houston, Texas 77251-1892

Abstract

We present a new method for solving Banning's alias-free flow-insensitive side-effect analysis problem. The algorithm employs a new data structure, called the *binding multi-graph*, along with depth-first search to achieve a running time that is linear in the size of the call multi-graph of the program. This method can be extended to produce fast algorithms for data-flow problems with more complex lattice structures.

1. Introduction

Interprocedural analysis of the side effects of subroutine invocation has been widely discussed in the literature [Spil 71, Alle 74, Bart 78, Bann 79, Rose 79, Myer 80, CoKe 84, Burk 84, BuCy 86, CaRy 86]. Banning has identified the important component problem of alias-free flow-insensitive side-effect analysis [Bann 79]. The fastest previous technique for solving this problem, which we call the *swift* algorithm, requires $O(NE\alpha(E,N))$ operations for instances of the problem with reducible call graphs [Coop 83, CoKe 84, CoKe 87a]. Here, E is the number of call sites, N is the number of procedures, and α is the functional inverse of Ackermann's function. The fundamental insight underlying the swift algorithm is that the problem can be subdivided into two subproblems: the side effects to parameters passed by reference and the side effects to variables passed as global variables. Each of these subproblems can then be solved using algorithms adapted from single-procedure data-flow analysis.[1]

In this paper we improve on the swift algorithm by presenting new algorithms for each of the two subproblems.

a) To solve for side effects to reference parameters, we use a graph of the parameter binding structure in a program, called the *binding multi-graph*. This approach yields a simple algorithm that takes $O(N+E)$ time in the worst case, assuming that the average number of parameters at procedures and call sites is bounded from above by a small constant.

b) Side effects to global variables can be determined by an algorithm that employs depth-first search to produce an answer in $O(N+E)$ bit-vector steps. It should be noted that bit vectors for interprocedural analysis will be exceedingly long. In fact, it is reasonable to assume that the number of global variables will grow linearly with the size of the program. Hence, the overall complexity of the resulting algorithm is $O(N^2+NE)$, although bit vectors can be used to speed up the analysis by a constant factor.

This paper divides into five major sections. Section 2 introduces the problem of interprocedural side-effect analysis and describes the partitioning. Section 3 introduces the binding multi-graph and shows how it can be used in an efficient algorithm for side-effect analysis. Section 4 describes the linear-time algorithm for analysis of side effects to global variables. Section 5 explains how to combine the results of the two problems to produce a solution for the original problem. Finally, section 6 shows how to extend this algorithm to handle the analysis of side effects to subsections of array variables. Throughout the paper, we will use the MOD problem as our example. The USE problem has an analogous solution.

2. The Problem

To determine the safety of applying an optimizing transformation, compilers examine the flow of values inside a procedure. Calls to external procedures present a difficulty for this type of analysis; if the compiler has no knowledge about the called procedure, it must assume that the called procedure both uses and modifies the value of every variable it can see. In practice, the called procedure typically modifies only a fraction of these variables. In a language like FORTRAN, where programmers use large numbers of global variables, the difference between assumption and reality is important. Thus, many authors have proposed that the compiler collect and use more precise information about the actual side effects of procedure calls. This sort of information should lead to improved

† This work has been supported by the NSF and IBM.

[1] The formulation of the decomposition presented in Cooper's dissertation and our SIGPLAN '84 paper contains a significant error. However, there are several corrections that both fix the problem and retain the time bound [Carr 87, CoKe 87a, CoKe 87b, Ryde 87]. The decomposition presented here is based on our own correction and revision of the SIGPLAN '84 paper [CoKe 87a, CoKe 87b].

optimization.

Specifically, the compiler should determine, for each call site, which variables can have their values modified by its execution and which variables can have their values used by its execution. To represent this information concisely, we annotate each call site s in the program with two sets, MOD(s) and USE(s), defined as follows. For a call site s and a variable v:

$v \in$ MOD(s) $\leftrightarrow$ executing s might change the value of v
$v \in$ USE(s) $\leftrightarrow$ executing s might use the value of v

We are interested in solving *flow-insensitive* versions of these problems. A flow-insensitive analysis concludes that a procedure call has a side effect, like $v \in$ MOD(s), if that side effect can occur on *some* path through the called procedure or any procedure that it, in turn, invokes. In other words, it ignores intraprocedural control structures. By contrast, a *flow-sensitive* analysis would conclude that the call has the side effect if and only if the analyzer can determine that the side effect occurs on *every* path through the called procedure and all procedures that it, in turn, calls.

A classical formulation of the flow-insensitive MOD problem, along the lines of Banning's work, makes an excellent detailed introduction to the problem [Bann 79].[2] Rather than compute MOD sets directly, Banning breaks the problem down into component parts. Aliasing is ignored until late in the computation; the method assumes that simple sets of alias pairs are available for each procedure. Next, define:

DMOD The computation of MOD(s) is complicated by aliasing effects. The treatment can be simplified by first computing DMOD(s), the set of variables that may be modified by execution of s, ignoring any aliasing effects in the procedure containing s, and factoring aliasing in later. In other words, MOD(s) can be computed by adding to DMOD(s) any variable that may be aliased to a member of DMOD(s). We call DMOD(s) the *directly modified set* for s.

GMOD The problem can be further simplified by observing that computing DMOD for any call site is easy once we determine, for each procedure p in the program, a set GMOD(p) that contains all variables that may be modified as the result of an invocation of p. We call GMOD(p) the *generalized modification set* for p.[3] Once it is computed, DMOD for any call site that invokes p can be computed by identifying the variables known at the call site that are bound by the call to variables in GMOD(p).

[2] This formulation is based on Banning's, but with different notation.

[3] In Banning's formulation, the GMOD set for the main program is empty by definition, since it cannot be invoked at a call site. We consider this an implementation detail and allow GMOD for the main program to be non-empty because it makes the formulation more natural.

The virtue of these observations is that GMOD(p) can be formulated as the solution to a system of data-flow equations on the call graph. To introduce this formulation, we need some more definitions.

LOCAL For a procedure p, LOCAL(p) contains the names of all variables declared in p.

LMOD For a statement s, LMOD(s) contains those variables that might be modified by an execution of s, exclusive of any procedure calls in s. We call LMOD(s) the *locally modified set* for s.

IMOD For a procedure p, IMOD(p) contains those variables that might be modified by an execution of p, exclusive of any procedure calls in p. We call IMOD(p) the *initially modified set* for p. Note that

$$\text{IMOD}(p) = \bigcup_{s \in p} \text{LMOD}(s).$$

We are now ready to introduce the system of equations for GMOD(p).

$$\text{GMOD}(p) = \text{IMOD}(p) \cup [\bigcup_{e=(p,q)} b_e(\text{GMOD}(q))] \quad (1)$$

Here, b_e is a function that maps names from q into names from p according to the name scoping and parameter binding that happens at the call site $e = (p,q)$. We call $b_e(x)$ the projection of x under the binding of e. It should be noted that b_e factors out all variables that are local to q and maps the formal parameters of q to the actual parameters at the call site. A similar system of equations can be used to define the USE computation.

Once GMOD(p) is known for each p, the DMOD set for a statement s can be computed by the following formula.

$$\text{DMOD}(s) = \text{LMOD}(s) \cup [\bigcup_{e=(p,q) \in s} b_e(\text{GMOD}(q))] \quad (2)$$

DMOD(s) contains those variables that are modified locally in s plus any variables that are modified as a result of executing any procedure calls contained in s. Thus, if s doesn't contain any procedure calls, DMOD(s) is identical to LMOD(s). If it does contain procedure calls, each such call contributes some projection of the GMOD set of the called procedure.

These equations are sufficiently complex that data-flow frameworks for their direct solution will not achieve the fast time bounds with any of the standard algorithms from global data-flow analysis [CoKe 87b]. To improve on the time bound, the *swift* algorithm relies on one central insight — we can decompose the problem into two subproblems: solving for effects due to reference parameter passing and solving for effects due to global variables.

Let us define IMOD$^+(p)$ to be the set of all variables that are either directly modified in p or passed by reference to another procedure and modified as a side effect of the invocation of that procedure. In other words, IMOD$^+(p)$ contains IMOD(p) along with all variables modified in p through side effects to reference parameters. If we can compute IMOD$^+(p)$ for each procedure p in the program, then we can reduce the problem of computing GMOD(p) to the solution of a system of equations analogous to equation (1).

$$\text{GMOD}(p) = \text{IMOD}^+(p) \cup [\bigcup_{e=(p,q)} b_e(\text{GMOD}(q))] \quad (3)$$

However, since we now have already solved for the effects of reference formal parameters, the function b_e takes on a particularly simple form. If procedure p calls procedure q, b_e needs to model modifications to variables that are extant after q returns. Clearly this means everything that is not local to q, because all of the local variables of q are deallocated on return[4]. Hence, equation (3) reduces to

$$\text{GMOD}(p) = \text{IMOD}^+(p) \cup [\bigcup_{e=(p,q)} (\text{GMOD}(q) \cap \overline{\text{LOCAL}(q)})]. \quad (4)$$

This system is trivially *rapid*, so that both the iterative algorithm and the Graham-Wegman algorithm will achieve their fast time bounds on an instance of the problem [KaUl 76, GrWe 76].

Thus, we have reduced the problem to the computation of IMOD$^+$. To do this, we further decompose the problem by introducing a new set RMOD(p) that contains all formal parameters to p that are modified as a side effect of invoking p. If we can compute this set for each procedure in the program, then IMOD$^+(p)$ can be computed by the following equation:

$$\text{IMOD}^+(p) = \text{IMOD}(p) \cup [\bigcup_{e=(p,q)} b_e(\text{RMOD}(q))] \quad (5)$$

where the function b_e is restricted to mappings arising from actual-to-formal parameter bindings. The problem, then, becomes one of computing RMOD(p) efficiently.[5]

In our previous work, we showed how to reduce the reference formal parameter problem to a *single source path expression problem* that can be solved using Tarjan's algorithm [Tarj 81a, Tarj 81b]. If we define c_P to be the maximum number of formal parameters in any single procedure and assume that c_P is independent of program size, then this algorithm requires $O(E\alpha(E,N))$ bit vector steps for reducible call graphs. We note that the analysis of side effects to global variables can also be performed in $O(E\alpha(E,N))$ bit vector steps using the same algorithm, although the bit vectors are much longer.

This paper presents linear-time algorithms for each of the two subproblems. Neither algorithm relies on the assumption of reducibility.

3. Reference Formal Parameter Problem

To solve the MOD problem for reference formal parameters, the swift algorithm solved a forward problem over the program's call multi-graph to compute summaries of parameter binding relationships and then used these summaries to produce MOD information. To simplify the reference formal parameter subproblem, we need to introduce a slightly different graph, the *binding multi-graph*. This is a simplification of the graph used in our algorithms for interprocedural constant propagation [CCKT 86, Torc 85].

3.1. The Binding Multi-Graph

The program's binding multi-graph, $\beta = (N_\beta, E_\beta)$, represents interactions between formal parameters. The nodes of β uniquely represent the formal parameters of the various procedures in the program. We denote them by the name of the procedure and the specific parameter's ordinal position, so that the third formal parameter for procedure p is written fp_p^3. Edges in E_β represent individual *binding events*. If p calls q from some call site s and fp_p^3 gets bound to fp_q^1 at s, then there is an edge $(fp_p^3, fp_q^1) \in E_\beta$.[6] Thus, a call site that passes only local variables as actual parameters generates no edges in E_β. Since p can call q several times, binding fp_p^3 to fp_q^1 at each call site, β may be a multi-graph. Because β reflects the pattern of binding chains in the program, it will almost certainly consist of a number of disjoint components.

How large is β? Since the complexity of data-flow algorithms is usually stated in terms of the size of the underlying graph, this issue is crucial to our later complexity analysis. The important comparison is to the program's call multi-graph, $C = (N_C, E_C)$. C contains a node for each procedure and an edge for each call site. Let μ_f be the average number of formal parameters, taken over all procedures in the program, and μ_a be the average number of actual parameters, taken over all call sites in the program. Now,

[4] In a block structured language like Pascal, all of the variables that are not local to q or some procedure defined in q are visible from within p. However, in Fortran a global variable modified by q may not be visible in p; nevertheless, it should be included in GMOD(p).

[5] The decomposition and its correctness are discussed in a pair of papers [CoKe 87a, CoKe 87b].

[6] We use fp_p^i interchangeably to name both the formal parameter and the node in N_β representing it. Similarly, we use set names like N_β to name both the set and its cardinality. In all cases, the meaning should be clear from the context.

we can clearly relate the size of C and β: $N_\beta \leq \mu_f N_C$ and $E_\beta \leq \mu_a E_C$. It is reasonable to assume that both μ_a and μ_f are independent of the growth in the program's size. In practice, programmers don't write ever longer parameter lists as the program grows — most of these interfaces are fixed at design time. Thus, we may assume that these quantities are bounded from above by a small constant k, so that $k \geq max(\mu_f, \mu_a)$, then β is only larger than C by a factor of k. Note that $k \leq c_P$.

The binding multi-graph can be constructed in time linearly proportional to its size by simply visiting each of the call sites in the original call graph. The construction need not represent a node unless it is the endpoint of an edge in E_β. Thus, $2E_\beta \geq N_\beta$, everywhere.

Practically, we expect k to be small. While there may exist individual procedures with large numbers of parameters, the averages μ_a and μ_f, taken across the whole program, should remain reasonably small. Furthermore, the fact that the graph need only represent those nodes that have at least one edge associated with them will have a moderating effect on β's size.

3.2. Using β to find RMOD

To solve the reference parameter problem, we must compute, for each procedure p, a set RMOD(p) that contains those fp_p^i's that may be modified by an execution of p. RMOD(p) is the contribution of p's reference formal parameters to GMOD(p). We can compute these sets directly, using β. With each node $fp_p^i \in N_\beta$ we associate two values that are the analogs, on β, of sets associated with the nodes in C. The first, IMOD(fp_p^i), gets the value *true* if and only if fp_p^i is modified locally in p. That is, IMOD(fp_p^i) is *true* if and only if $fp_p^i \in$ IMOD(p). Otherwise, it gets the value *false*. The second, RMOD(fp_p^i), gets initialized to *false*, $\forall$ $fp_p^i \in N_\beta$. Now, the RMOD problem can be posed as the solution to the following system of data-flow equations:

$$\text{RMOD}(m) = \text{IMOD}(m) \bigvee \left(\bigvee_{c=(m,n)\in E_\beta} \text{RMOD}(n) \right) \qquad (6)$$

where $\bigvee$ is the *logical or* operator. This set of equations has the interesting property that its solution is identical at every node within a strongly connected region. To solve this set of equations, we can use the simple algorithm shown in Figure 1. Since each of the steps in this algorithm takes no more than $O(N_\beta+E_\beta)$ time, the whole process has that time bound.

To understand the effectiveness of this method, it is important to compare it to the swift algorithm. The time bound for the swift algorithm is in terms of bit vector operations, where each bit vector is as long as the total number of reference formal parameters in the program, or

(1) Find the strongly connected components (SCC's) of β.

(2) Replace each SCC with a representer node n, setting IMOD(n) to the *logical or* of all the IMOD sets of the nodes in the SCC. Set RMOD(n) to *false*.

(3) Traverse the derived graph from leaves to roots, applying equation (6).

(4) For each SCC, set the RMOD set for each node in the SCC to the value of the RMOD set for its representer node.

Figure 1 — Solving for RMOD

N_β bits. In data-flow analysis of a single procedure, it is commonly assumed that the bit-vector length does not grow appreciably with the size of the procedure. In interprocedural analysis, however, we expect bit-vectors to grow linearly as the size of the program, since programs are typically built by adding more procedures (hence, more parameters), rather than by increasing the size of each procedure. Since the swift algorithm requires $O(E_C\alpha(E_C,N_C))$ bit-vector operations, its complexity is $O(N_\beta E_C\alpha(E_C,N_C))$ under this assumption.

By comparison, the method based on the binding multi-graph takes $O(E_\beta) = O(kE_C)$ simple logical steps, where k is the constant upper bound on the *average* number of parameters, as defined in Section 3.1. Typically, this will be a small constant (less than 20), no matter how large the program grows. Hence, the new algorithm can be said to be an order of magnitude faster than the swift algorithm.

The new method might be viewed as the analog for interprocedural analysis of Zadeck's PVT algorithm applied to a backward data-flow problem in a single procedure [Zade 84]. However, in Zadeck's method the algorithm is applied once for each variable or cluster of variables; for our method, a single application to β suffices. We have gained significant leverage by changing graphs.

3.3. Lexical Scoping

This method handles the two-level name scoping of C or FORTRAN. However, languages like Pascal, which permit nested declaration of procedures, present a special problem because the method determines effects to global variables *after* it determines effects to formal parameters. In a language with nested procedure declarations, a local variable for one routine is global to procedures declared within the body of that routine. Thus, nesting can affect the computation of RMOD in two ways:

1) IMOD(p) must reflect modifications to a local variable v that happen inside a nested procedure, where v is a global variable.
2) One of p's formal parameters may be used as an actual parameter at a call site within some nested procedure q. This binding must be reflected in the construction of β.

Fortunately, both these problems are easily solved. Assume that every procedure in the program is reachable by some call chain. If this is not the case, a linear-time algorithm that eliminates unreachable procedures can be invoked. Now any procedure q nested within procedure p is reachable by a call chain starting at p because no procedure outside of p can invoke q directly, since it is not visible outside of p. Hence, if q is reachable, it is reachable from p. This means that if p is invoked, we must assume that q may be invoked.

Given these observations and the flow-insensitive nature of the computation, the first problem above is solved by treating the bodies of procedures nested in p as extensions of the body of p. This is no different than assuming that each branch at a conditional statement is possible.

We extend the IMOD(p) sets to include variables that are visible within p (global or local to p) and are directly modified within the body of p or passed as globals to some procedure whose declaration is nested within p and directly modified there. If we let $Nest(p)$ be the set of procedures declared in p, we can formulate the following definition for IMOD(p).

$$\text{IMOD}(p) = \bigcup_{s \in p} \text{LMOD}(s) \cup \left[\bigcup_{q \in Nest(p)} (\text{IMOD}(q) \cap \overline{\text{LOCAL}(q)}) \right]$$

The IMOD sets can then be computed in a bottom up fashion — first for the most deeply-nested procedures and then for the procedures containing those. This computation is linear in the size of the program. The redefinition of IMOD leads to a corresponding redefinition of IMOD$^+$.

The second problem, a formal parameter of p used as an actual parameter at some call site inside a nested procedure q, is easy to handle. Whenever the graph constructor encounters a formal parameter of p being passed as an actual parameter at some call site in q, where q is lexically nested within p, it must add the appropriate edge from p's formal parameter to the corresponding formal of the called procedure. A careful initialization of the basic data structures used to construct β will ensure this.

4. Global Variable Problem

Equation (4) is a particularly simple system of equations. However, it becomes even simpler if we are dealing with a language, like C or FORTRAN, where all variables are partitioned into two classes: global and local. In that case, the determination of which variables are local and which are global is independent of the procedure being invoked. In other words, GMOD for a particular procedure p is simply IMOD$^+(p)$ augmented by those global variables that are modified in some procedure reachable by a call chain from p. This suggests that we might view the problem as a generalization of the reachability problem and adapt depth-first search to produce a solution.

Figure 2 presents the algorithm for computing GMOD sets from the IMOD$^+$ sets. The algorithm is adapted directly from Tarjan's strongly-connected components algorithm [Tarj 72]. The basic idea is to compute an initial approximation to GMOD[p] that includes all variables that may be modified as a side effect of call chains that include tree edges, forward edges, or cross edges to nodes that are in different strongly-connected components of the depth-first search tree for the call graph. Whenever the root of a strongly-connected component is found, its GMOD set represents all side effects that can occur in procedures within the strongly-connected component, since all procedures in such a component are reachable from the root by tree edges. Thus, it is correct to augment the GMOD set for each member of the strongly-connected component by the set of variables in GMOD[*root*] that are not local to the root.

The remainder of the section presents a formal proof of the correctness of this algorithm. The proof is based upon the proof of Tarjan's algorithm. In fact, the only substantive additions to that algorithm are lines 8, 17 and 22, which represent partial applications of equation (4). We need only show that these applications have the effect of correctly computing GMOD for each node.

We say that a strongly-connected component is *closed* when, in line 19, its *root*, or member with lowest depth-first number, is found and all its members are popped off the stack. We will show that whenever a strongly-connected component is closed, the GMOD sets for each of its members is correctly computed. Tarjan adopted the convention that each vertex could reach itself by the empty path, so even if some vertex is cannot reach itself by an explicit sequence of edges, it is still a member of the strongly-connected component containing only itself. Hence, proving this result will establish correctness of the algorithm.

We begin by establishing some preliminary properties of the algorithm.

Lemma 1. *If there is an edge $e = (p,q)$ from a member p of strongly-connected component c_1 to a member q of a second strongly-connected component c_2, then c_2 must be closed before c_1.*

Proof. Suppose this is not the case — that is, suppose that c_1 is closed before c_2. There are two cases to consider.

```
1       procedure findgmod;
2           integer      dfn[N], lowlink[N], nextdfn, p, q, d, l,
3                        GMOD[N], IMOD+[N], LOCAL[N];
4           integer stack   Stack;
5           procedure search(p);
6               integer p, q;
7               dfn[p] := nextdfn;  nextdfn := nextdfn + 1;
8               GMOD[p] := IMOD+[p];
9               lowlink[p] := dfn[p];
10              push p on Stack;
11              foreach q adjacent to p do begin
12                  if dfn[q] = 0 then begin /* tree edge */
13                      search(q);
14                      lowlink[p] := min (lowlink[p], lowlink[q]);
15                  end;
14                  if dfn[q]<dfn[p] and q∈Stack then /* cross or back edge, same scc */
15                      lowlink[p] := min (dfn[q], lowlink[p]);
16                  else /* apply equation (4) */
17                      GMOD[p] := GMOD[p] ∪ (GMOD[q] ∩ ¬LOCAL[q]);
18              end;
                /* test for the root of a strong component */
19              if lowlink[p] = dfn[p] then begin
                    /* adjust GMOD sets for each member of the scc */
20                  repeat begin
21                      pop u from Stack;
22                      GMOD[u] := GMOD[u] ∪ (GMOD[p] ∩ ¬LOCAL[p]);
23                  end
24                  until u = p;
25              end
26          end /* search /* ;
            /* assume that IMOD+ and LOCAL have been initialized */
27          nextdfn := 1;  dfn[*] := 0,  Stack := ∅;
28          search(1);  /* root = 1 */
29      end /* findgmod */
```

Figure 2 — One-level global side effect algorithm

1) The root of c_2 is put on the stack after the root of c_1. If this happens, it must be the case that c_1 is closed before the root of c_2 is stacked, since the algorithm dictates that if two roots are on the stack at the same time, the component corresponding to the shallower root will be closed first. But c_1 cannot be closed until all the nodes of c_2 are visited because of the edge between $p \in c_1$ and $q \in c_2$. Depth-first search will explore all paths from p, including the path into c_2, before it returns. Hence the root of c_2 must be visited and closed before the search returns to p and this must take place before c_1 can be closed.

2) The root of c_2 is put on the stack before the root of c_1. Since it must remain on the stack until c_2 is closed, it must still be on the stack when c_1 is closed, because we have assumed that c_1 is closed before c_2. It is a fundamental property of depth first search that there is a path from any given node on the stack to all nodes that are stacked on top of it. Thus, there must be a path from the root of c_2 to every node in c_1. In particular, there must be a path from the root of c_2 to p, from which there is an edge to $q \in c_2$. Since, c_2 is strongly connected, there is a path from q to every node of c_2, including the root. We conclude, therefore, that c_1 and c_2 must be the same strongly-connected component, a contradiction. **Q.E.D.**

Now consider the value of GMOD[p] on exit from the loop in lines 11-18. It is an initial approximation to GMOD[p] that is a subset of the correct GMOD, since lines 8 and 17 implement equation (4). The following lemma establishes an important property of that approximation.

Lemma 2. *If q can be reached from p by a possibly empty path consisting exclusively of tree edges, then*

$$\text{GMOD}[p] \supseteq \text{GMOD}[q] \cap \overline{\text{LOCAL}[q]}. \tag{7}$$

Proof. The proof is by induction on the order of visits by depth-first search. First, note that if GLOBAL is the set of all global variables in the program, then

$$\text{GMOD}[q] \cap \overline{\text{LOCAL}[q]} = \text{GMOD}[q] \cap \text{GLOBAL} \tag{8}$$

since the only local variables that can be modified as a side effect of the invocation of a procedure q are its own. If there are no tree edges out of p then equation (7) holds vacuously, since the only possible q is p itself. In particular, it holds for the node with the greatest depth-first number. Suppose that the lemma holds for all nodes with a greater depth-first number than p. Let x be a variable modified in some q reachable from p by tree edges. If $q = p$, equation (7) holds trivially, so assume there is at least one edge in the path to q and let (p,u) be the first edge. Since u has a greater depth first number than p, we have by the induction hypothesis

$$\text{GMOD}[u] \supseteq \text{GMOD}[q] \cap \overline{\text{LOCAL}[q]} = \text{GMOD}[q] \cap \text{GLOBAL}.$$

By line 17, we conclude

$$\begin{aligned} \text{GMOD}[p] &\supseteq \text{GMOD}[u] \cap \overline{\text{LOCAL}[u]} = \text{GMOD}[u] \cap \text{GLOBAL} \\ &\supseteq \text{GMOD}[q] \cap \text{GLOBAL} = \text{GMOD}[q] \cap \overline{\text{LOCAL}[q]}, \end{aligned}$$

the desired result. **Q.E.D.**

Theorem 1. *Given that the* IMOD$^+$ *sets are properly initialized, algorithm* findgmod *correctly computes* GMOD(p) *for each procedure p.*

Proof. The proof of the theorem is by induction on the order in which components are closed. We assume as an induction hypothesis that GMOD sets have been correctly computed for each strongly-connected component closed before the current one, which we call c. This hypothesis is vacuously true when c is the first component to be closed.

We claim that when the root r of c is reached in statement 19, GMOD[r] is the correct GMOD set for r — that is, it contains all the global variables that can be modified, either directly or as a side effect to a reference formal parameter, within any procedure reachable from r. Suppose there exists a global variable x, not a member of GMOD[r], for which there is a (possibly empty) path from r to a procedure in which x is modified. If x is modified in r itself, it will be reflected in GMOD[r] by virtue of the initialization of that set to IMOD$^+$[r] in line 8. So assume there is a non-empty path from r to a procedure in which x is modified. If the path contains only tree edges, x must be in GMOD[r] by equations (7) and (8). If the path contains forward edges, x will still be in GMOD[r], since any q reachable from r by a path containing only tree edges and forward edges can be reached from r by a path containing only tree edges.

Thus, we must assume that the path contains a cross edge to a different strongly-connected component. The component must be different, because there is a path consisting exclusively of tree edges from r to every member of c. Let (p,q) be the first such edge. By Lemma 1, the component containing q must already be closed, so the induction hypothesis applies and we must have $x \in$ GMOD[q]. But this means that $x \in$ GMOD[p] since every global member of GMOD[q] is added to GMOD[p] when line 17 is executed for (p,q). Since p is reachable from r by tree edges, we have by Lemma 2 and equation (8)

$$x \in \text{GMOD}[p] \cap \text{GLOBAL} = \text{GMOD}[p] \cap \overline{\text{LOCAL}[p]} \subseteq \text{GMOD}[r].$$

The contradiction establishes the correctness of GMOD[r].

The theorem now follows from the observation that the set of global variables modified at any member of a strongly-connected component must be the same as the set of such variables modified at the root. In other words, for each member q of the component with root r,

$$\begin{aligned} \text{GMOD}[q] \cap \text{GLOBAL} &= \text{GMOD}[r] \cap \text{GLOBAL} \\ &= \text{GMOD}[r] \cap \overline{\text{LOCAL}[r]} \end{aligned}$$

by equation (8). Hence, line 22 correctly adjusts the GMOD set for each member to include all global variables in GMOD for the root. **Q.E.D.**

Theorem 2. *If sets are represented as bit vectors, algorithm* findgmod *requires $O(E_C+N_C)$ bit-vector steps.*

Proof. Line 17 is executed no more than once for each edge and line 22 is executed no more than once for each vertex. **Q.E.D.**.

The technique can be extended to languages in which procedures can be declared at multiple nesting levels by the simple device of simultaneously solving the problem for each nesting level. That is, suppose we number the procedure declaration nesting levels 0 through d_P, where level 0 is the nesting level of the main program and d_P is the maximum level at which any procedure in the program is declared. If $d_P = 1$, the problem reduces to the simple global-local problem discussed above.

When $d_P > 1$, we can simultaneously develop the solutions to problems numbered 1 through d_P, where the solution to problem i includes effects to global variables for call chains that never invoke a procedure at a nesting level shallower than i. That is, the i^{th} problem is defined on a graph in which all edges representing calls to procedures declared at levels shallower than i are ignored.

It is easy to solve all these problems in $O(d_P(E_C+N_C))$ bit-vector steps by simply repeating the algorithm from figure 2 for each level i on the associated graph. However, by maintaining a vector of *lowlink* values, one for each problem, we can eliminate d_P as a multiplier of E_C. The key insight is that a strongly-connected region that includes no procedure at a nesting level shallower than i will be a proper subset of the maximal strongly-connected region that includes all the same nodes but may include procedures at a higher nesting level. This means that the *lowlink* vectors will be ordered in value, with the *lowlink* for the problem at level i less than or equal to the lowlink for the problem at level $i+1$. Thus, in the loop at line 11 in Figure 2, the algorithm can simply adjust the *lowlink* corresponding to the nesting level of the called procedure. After exiting the loop, but before testing for a strongly-connected region, the *lowlink* vector must be corrected by insuring that values from lower nesting levels are propagated to higher nesting levels where appropriate, a step that takes time proportional to d_P. If we maintain parallel stacks, the lines between 19 and 25 are executed at most once for each nesting level, so the time required is proportional to $d_P N_C$.

The result is an algorithm that solves all d_P problems and computes the union of these solutions in $O(E_C+d_P N_C)$ bit vector steps. Since d_P is likely to be bounded by a small constant independent of program size, this is effectively $O(E_C+N_C)$ bit vector steps. Assuming that the bit vectors are of length $O(N_C)$, the total time required by the global analysis phase is $O(E_C N_C+N_C^2)$.

5. Computing MOD

Given GMOD for each procedure, computing MOD sets for the call sites is a two-step process.

(1) For each call site s, compute DMOD(s) by applying equation (2).

(2) To obtain MOD(s) from DMOD(s), extend DMOD(s) to account for aliases. That is, if we have a set ALIAS(p) containing the alias pairs that can hold on entry to p, $s \in p$, then

$\forall\, x \in$ DMOD(s), if $\exists\, <x,y> \in$ ALIAS(p), add y to MOD(s).

Step (1) takes $O(N_C E_C)$ time. Step (2) takes time linear in the size of DMOD(s) and ALIAS(p). While ALIAS(p) can grow to be large, programs with complex aliasing patterns are difficult to write and understand. Any algorithm that computes summary information must deal with the aliases; it will require at least time linear in the number of aliases, as we do.[7] In the absence of aliasing, the entire process requires $O(N_C(E_C+N_C))$ time.[8] When a large number of aliases exist, our method, like any other method, will require time proportional to the size of the ALIAS sets.

6. Regular Section Analysis

Our experience with using interprocedural summary information in a working system for detecting parallelism has shown that the granularity of conventional summary information is too coarse to allow effective detection of parallelism in loops that contain call sites [CaKe 87]. The problem lies with the treatment of whole arrays. The standard framework for interprocedural analysis treats formal parameters as unitary objects. Hence, if the formal parameter is an array, side effects that are restricted to a portion of the array will be reported as having affected the whole array. In other words, these methods are able to determine whether an array is modified somewhere, but not whether it is modified in only a single column or row. This limitation is disastrous for parallelization because the most effective way to parallelize a loop is through data decomposition, in which each parallel processor works on a different subsection of a given array.

Callahan and Kennedy have proposed a technique, called *regular section analysis*, to solve this problem

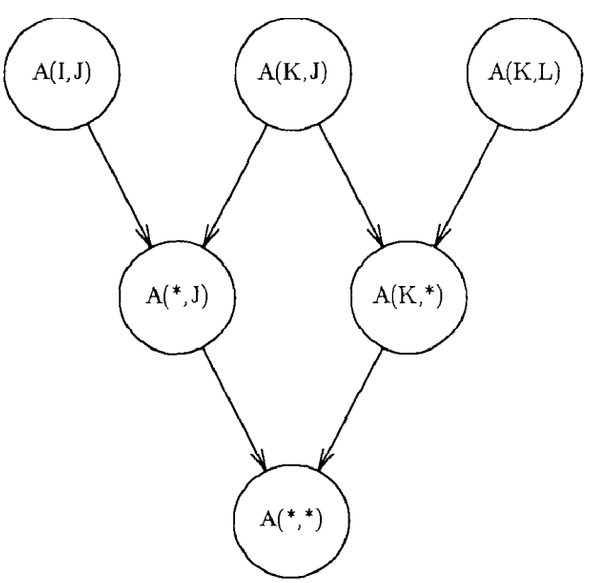

Figure 3 — Simple regular section lattice

[7] The stated time bound for the swift algorithm and other algorithms for this problem ignore the term for aliases. We will continue this practice.

[8] Recall that the sizes of C and β are related by a small constant k.

[CaKe 87].[9] The basic idea is to replace the single-bit representation of side effects with a richer lattice that permits the representation of subregions. A *regular section* is a subregion of an array that has an exact representation in the given lattice.

Perhaps the best way to illustrate this idea is by an example. Figure 3 displays a lattice of reference patterns to array A in which the regular sections are single elements, whole rows, whole columns and the whole array. Note that I, J, and K are arbitrary symbolic input parameters to the call.

Callahan and Kennedy point out that a variety of algorithms can be accommodated in the regular section framework—these algorithms would differ only in the cost of the representation of lattice elements, the cost of determining whether two lattice elements represent an intersecting subsection (used for dependence testing), the expense of the *meet* operation and the depth of the lattice. They also claim that most standard bit-vector algorithms can be extended naturally to deal with regular sections, although it is not immediately clear how to do this with the swift algorithm.

The approach proposed in this paper extends very naturally to lattice elements. Clearly, the bit vector technique for solving the global variable problem can be directly extended to vectors of lattice elements. Extending the algorithm for the reference formal parameter problem is not so straightforward, however. The principle complication is that formal parameter arrays are often bound to *subsections* of actual parameter arrays. The implication is that, during the analysis, the regular section describing access to a *formal* parameter must be mapped to a regular section describing access to the *actual* parameter by a function that may not be the identity function. Hence, the simple trick to handle cycles in the binding multi-graph will no longer work.

Formally, each edge e in the binding multi-graph must be annotated with a function g_e that can be used to map a regular section at its sink to one at its source. Before we can discuss the impact that this has on the underlying algorithms, we need to describe reasonable properties for the functions g.

- First, the functions can be extended to functions on paths by using composition. In other words, if $p=e_1e_2\cdots e_n$, then $g_p=g_{e_1}g_{e_2}\cdots g_{e_n}$.
- The functions can be extended to sets of paths by using the lattice meet operation. That is, for a regular section x, $g_{p_1\cup p_2}(x)=g_{p_1}(x)\wedge g_{p_2}(x)$.
- Because of the nature of parameter passing in most languages, it is almost always the case that *subsections* of the actual parameter are passed to the formal parameter[10]. This means that, around a cycle of the binding multi-graph, the effect of the propagation functions is to *restrict* the portion of the array that is involved in a side effect. More formally, if p is a cyclic path leading from a formal parameter back to itself through a sequence of calls, then $g_p(x)\wedge x = x$.

This last observation is critical because, if we assume it as a restriction, we can view the regular section problem as a data-flow framework with the following system of equations.

$$rsd(fp_1) = lrsd(fp_1) \wedge \bigwedge_{e=(fp_1,fp_2)\in E_\beta} g_e(rsd(fp_2))$$

Here, $rsd(x)$ is the regular section descriptor for the side effect to parameter x and $lrsd(x)$ is the regular section descriptor for the side effect due to local effects within the procedure where x is declared as a formal parameter (computable by local examination of a procedure). This framework is fast in the sense of Kam and Ullman [KaUl 76] and rapid in the sense of Graham and Wegman [GrWe 76]. Furthermore, it can be formulated as a *path problem* [Tarj 81a], so any of the efficient elimination techniques can be used to find a least fixed point.

For the sake of complexity analysis, let us consider using the most efficient known data-flow analysis method to solve this problem—Tarjan's fast elimination method based on path compression [Tarj 81b]. If the binding multi-graph is reducible, this approach takes time $O(E_\beta\,\alpha(E_\beta, N_\beta))$. Here, the time is roughly proportional to the number of meet operations. Since E_β and N_β are at most a small constant factor k larger than E_C and N_C, the complexity is $O(E_C\alpha(E_C,N_C))$. This is the same asymptotic complexity as the swift algorithm, although the meet operations may be more expensive.

One surprising fact about this algorithm is that the complexity does not depend on the depth of the lattice, a byproduct of the third assumption above. In a sense, one can view the third assumption as recognizing that most recursive algorithms that pass a parameter over a recursive call cycle to the same position are using a form of divide-and-conquer.

7. Conclusions

We have introduced a new approach to dealing with interprocedural side effects to reference parameters — through the use of the binding multi-graph. This technique permits solution of the parameter side effect problem in

[9] Callahan and Kennedy discuss regular section analysis in more depth, as well as other methods proposed by Burke and Cytron, and by Triolet, Irgoin, and Feautrier [BuCy 86, TrIF 86].

[10] FORTRAN can be an exception, but we view those cases as pathological even for FORTRAN.

linear time. The remaining problem of analyzing side effects to global variables in a language with no reference parameters has been shown to be solvable by an adaptation of Tarjan's strongly connected components algorithm. These techniques are also useful for analyzing more complex side effects, such as those to subsections of arrays. In each case, the time bound achieved is asymptotically the fastest known. We also expect these algorithms to be extremely fast in practice.

8. Acknowledgements

Throughout our work on this problem, we have benefitted from discussions with David Callahan, Linda Torczon, and Barbara Ryder. Fran Allen, Corky Cartwright, and Guy Steele all contributed with informed criticism and comments. To these people go our heartfelt thanks.

References

[Alle 74] F.E. Allen, "Interprocedural data flow analysis", *Proc. of the 1974 IFIPS Congress*, 1974.

[Bann 79] J.P. Banning, "An efficient way to find the side effects of procedure calls and the aliases of variables", *Proc. of the Sixth POPL*, Jan., 1979.

[Bart 78] J.M. Barth, "A practical interprocedural data flow analysis algorithm", *CACM* 21(9), Sept., 1978.

[Burk 84] M. Burke, "An interval analysis approach toward interprocedural data flow", Report RC 10640, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., July, 1984.

[BuCy 86] M. Burke and R. Cytron, "Interprocedural dependence analysis and parallelization", *Proc. of the SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), July, 1986.

[CCKT 86] D. Callahan, K.D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation", *Proc. of the SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), July, 1986.

[CaKe 87] D. Callahan and K. Kennedy, "Analysis of interprocedural side effects in a parallel programming environment", *Proc. of First Int'l Conference on Supercomputing*, Athens, Greece, June, 1987.

[CaRy 86] M.D. Carroll and B.G. Ryder, "An incremental algorithm for software analysis", *Proc. of the* SIGSOFT/SIGPLAN *Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices* 22(1), Jan., 1987.

[Carr 87] M.D. Carroll, "Dataflow update via attribute and dominator update," Ph.D. Dissertation, Rutgers University, 1987.

[Coop 83] K.D. Cooper, "Interprocedural data flow analysis in a programming environment", Ph.D. dissertation, Department of Mathematical Sciences, Rice University, April, 1983.

[CoKe 84] K.D. Cooper and K. Kennedy, "Efficient computation of flow insensitive interprocedural summary information", *Proc. of the SIGPLAN 84 Symposium on Compiler Construction, SIGPLAN Notices* 19(6), June, 1984.

[CoKe 87a] K.D. Cooper and K. Kennedy, "Efficient computation of flow-insensitive interprocedural summary information — a correction", TR87-60, Department of Computer Science, Rice University, Oct., 1987.

[CoKe 87b] K.D. Cooper and K. Kennedy, "Complexity of interprocedural side-effect analysis", TR87-61, Department of Computer Science, Rice University, Oct., 1987.

[GrWe 76] S. Graham and M. Wegman, "A fast and usually linear algorithm for global flow analysis", *JACM*, Jan., 1976.

[KaUl 76] J. Kam and J. Ullman, "Global data flow analysis and iterative algorithms", *JACM*, Jan., 1976.

[Myer 80] E. Myers, "A precise and efficient algorithm for determining existential summary data flow information", Technical Report CU-CS-175-80, Department of Computer Science, University of Colorado, March, 1980.

[Rose 79] B. Rosen, "Data flow analysis for procedural languages", *JACM* 26(2), April, 1979.

[Ryde 87] B. Ryder, private communication, July 31, 1987.

[Spil 71] T.C. Spillman, "Exposing side-effects in a PL/I optimizing compiler", *Proc. of the 1971 IFIPS Congress*, 1971.

[Tarj 72] R.E. Tarjan, "Depth-first search and linear graph algorithms", *SIAM J. Computing* 1(2), 1972.

[Tarj 81a] R.E. Tarjan, "A unified approach to path problems", *JACM* 28(3), July, 1981.

[Tarj 81b] R.E. Tarjan, "Fast algorithms for solving path problems", *JACM* 28(3), July, 1981.

[TrIF 86] R. Triolet, F. Irgoin, and P. Feautrier, "Direct parallelization of call statements", *Proc. of the SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), July, 1986.

[Torc 85] L. Torczon, "Compilation dependences in an ambitious optimizing compiler", Ph.D. dissertation, Department of Computer Science, Rice University, May, 1985.

[Zade 84] F.K. Zadeck, "Incremental data flow analysis in a structured program editor", *Proc. of the SIGPLAN '84 Symposium on Compiler Construction*, SIGPLAN Notices, 19(6), June 1984.